UiO : Department of Informatics
University of Oslo

Taming Redundant Data Bundling

Balancing fairness and latency for redundant bundling in TCP

Bendik Rønning Opstad

Master's Thesis Spring 2015

Taming Redundant Data Bundling

Bendik Rønning Opstad

Spring 2015

Abstract

The Internet is used by a vast range of different services with different requirements and needs. Along with the continous increase in both bandwidth capacity and users, we have seen a development in the later years of more and more latency dependent applications being used. The strict latency requirements of applications such as online gaming and IP-telephony (*Voice over IP* **(VoIP)**) differ greatly from traditional greedy applications like FTP, that seeks to achieve the best possible throughput. As all the network traffic share the same network resources, the underlying network has a difficult task in trying to balance the resources and the needs of different applications.

Being the most used protocol on the Internet, *Transmission Control Protocol* **(TCP)** is used for many different kinds of services. Due to its reliabillity guarantees of in-order data delivery and error detection, it is the first choice for many use cases where the investment in custom solutions, based on such as *User Datagram Protocol* **(UDP)**, cannot be justified.

Applications transfering time-dependent data, often send thin-stream traffic, characterized by small packet payload and high *inter-transmission times* **(ITTs)**. The major cause of high latencies in thin stream are lost packets, and how the mechanisms for retransmitting the lost data work (*Griwodz* and *Halvorsen* [2006, a1]).

Thin streams suffer from the fact that TCP has been tuned for greedy traffic, where the low ITT ensures that retransmissions are initiated faster, giving satisfactory results even for latency sensitive data such video streaming (*Wang* et al. [2008, a2]).

In this thesis we continue the work on a sender side TCP modification called *Redundant Data Bundling* **(RDB)** for the Linux kernel, that aims at improving the latencies for thin streams, without being unfair to competing network streams.

Acknowledgements

First I would like to thank my supervisors, Dr. Andreas Petlund, Dr. Pål Halvorsen, and Dr. Carsten Griwodz, for great guidance during the work on this thesis. They have all provided a wide range of useful feedback on the background work, as well as on the writing process of the thesis. Their understanding for how much time certain tasks may require is also greatly appreciated.

A special thanks goes to Jonas Sæther Markussen, for a long and fruitfull colaboration that served us both well. Also the countless discussions on various topics during our time here will be remembered.

Finally, I would like to thank my family and friends for their support.

Contents

Appendices

List of Figures

Chapter 2

Chapter 3

Chapter 4

Chapter 5

List of Tables

Chapter 2

Chapter 3

Appendix A

List of Equations

Chapter 2

Chapter 4

Chapter 5

List of Source code listings

Chapter 2

Chapter 4

Chapter 5

Appendix B

Appendix C

List of command examples

Chapter 5

Glossary

ACK latency is the time between a data segment is first sent onto the network, until an ACK for the data segment is received. xii, 4, 24, 46, 85, 86, 95, 98–100, 104, 107, 110–113, 116, 117, 119, 121, 122, 124–126, 135

acknowledgment is the term used for packets that acknowledge that certain data has been received. In this thesis we refer to the TCP's acknowledgment packets which are regular TCP packets with th ACK flag in the TCP header indicating the highest sequence number that has been received in order. xxix, 13, 21

additive-increase is a phase where the send rate is increased additively. 17, 21

application layer latency is the latency from the time a data chunk is sent from the application layer on the sender side to the data chunk is delivered to the application layer on the receiver side. 28, 59, 85, 86, 116, 135

application limited streams are network streams that are limited by the application and not the network. xxvi, 11, 12, 35, 53, 55, 56, 265, 267

congestion avoidance is a phase after the initial growth phase where the CWND is increased slowly. Standard TCP follows the AIMD paradigm to control the CWND. xxv, 16, 17, 56

congestion collapse is a condition in a packet switched network where the goodput is minimal or none-existing. 1, 16, 27

congestion control describes the mechanisms or algorithms used by a network protocol to control the send rate to avoid network congestion. xxix, 1, 4, 21

TCP BIC was the default CC algorithm in the Linux kernel from version 2.6.8 till 2.6.19 when it was replaced by TCP Cubic. 29

TCP Cubic is the default CC algorithm in the Linux kernel, introduced in version 2.6.19 (7e3801). xii, xxi, 19, 29, 39, 56, 74, 76, 116, 128

TCP New Reno is an improvement on TCP Reno, and was the default CC in the Linux kernel until v2.6.8 was released in August, 2004. xvii, xviii, 19, 29, 39, 56, 76, 82, 109, 115, 127, 265, 266, 281

TCP Nice is a CC algorithm implemented in the Linux kernel. "TCP–Nice is an experimental congestion control mechanism that uses less than it's fair share of bandwidth when there is congestion, much like nice does for CPU usage by processes in the Unix operating system." (*Mcdonald* and *Nelson* [2006, a3]). 82

TCP Reno is a CC algorithm named after the BSD version "4.3BSD-Reno" where it was first implemented. xxi, 16, 19

TCP Vegas is a CC algorithm that relies on how the RTT changes to control the CWND. 19

TCP Westwood is a CC algorithm that relies on changes in the throughput to control the CWND. 19

Congestion Window is the window specifying the amount of outstanding data a TCP sender host may have. xxix, 17

Eifel detection algorithm described both a flag-based, and a timestamp-based algorithm for unambiguously distinguish which data packet an incoming ACK is a response to. (*RFC3522*) 25, 61, 62

E-model is a standardized model for measuring the quality of speech. 8

exponential backoff is a mechanism where the RTO timer is increased exponentially, by being doubled on successive RTOs. 27, 31

fast recovery is a variation to the CC algorithms that handles the sending of new data after a fast retransmit. 17, 18, 20, 26, 29, 32

fast retransmit triggers a retransmission based on incoming dupACKs. xi, xxii, 13, 17, 18, 30, 32–35, 55, 113

first person shooter game is a game where the player controls an avatar in first person, moving around in a virtual world where the goal is to shoot other players. xxix, 10

flight size is the amount of outstanding data for a stream in the network (as defined by *RFC5681*). For segment based accounting it is natural to express this as the number of segments/packets that have not been ACKed. 13, 35

goodput is the useful part of the throughput, i.e. the data that has not already been transmitted. 4, 28, 95, 104–106, 118, 124

greedy stream is a network stream that aims to achieve the maximum possible throughput. Such streams are network limited, with the exception of scenarios where the sender is limited by such as local I/O. xi, xxvi, 7, 11, 14, 27, 28, 30, 46, 49, 53, 54, 56, 82, 91, 93, 94, 104, 105, 109, 115, 116, 120, 122

Head-of-line blocking is when packets in a queue are held or blocked until the first packet in the queue is processed. In a fifo-queue, any newer packets are held until the oldest packet can be processed and removed. For protocols such as TCP, which have an in-order guarantee to the application layer, any out-of-order packets are held on the receiver side until the sequence number gap is filled. xxx, 1

inter-arrival time xxx, 32

Internet Protocol Security is a mechanism for securing network traffic at the IP layer. xxx, 89

inter-transmission time i, xxx, 12, 13

Jain's fairness index is an equation for rating the fairness of network streams, most commonly by throughput. 28

Karn's algorithm is used to improve the accuracy of the SRTT by excluding the RTT measurements for retransmitted packets in the calculation. 24, 25

Linear Retransmission Timeout is a modification on TCP in the Linux kernel, suggested and implemented in *Petlund* [2009, a4] and *Evensen* [2008, a5]. It was introduced in the official Linux kernel version 2.6.34 (*Petlund* [2010a, b1]) as `tcp_thin_linear_timeouts`. xviii, xxvi, xxx, 30

Linux CC framework is the pluggable module system for CC implementations in the Linux kernel. xii, xviii, 39, 56, 66, 77–79, 82, 83, 274

Linux kernel is the kernel we have implemented and tested RDBv2 on. Unless otherwise specified, we refer to the kernel version 3.16 i, xi, xii, xvii, xxi–xxvii, 2–5, 16, 19, 20, 23, 25, 27, 30, 31, 33, 35, 38, 39, 41, 43, 45, 53, 57–59, 65, 76, 77, 82, 88, 91, 94, 103, 104, 109, 115, 127, 133, 134, 266, 267

Massively multiplayer online role-playing game is a type of online game that combines the role-playing genre with the support for a large number of simultaneous players, also known as massively multiplayer online game. xxx, 10

- **Age of Conan** 10
- **Anarchy Online** 10, 13
- **World of Warcraft** is an MMORGP created by Blizzard Entertainment released in 2004. xxxii, 10

minimum RTT is the minimum registered RTT within a certain interval. 60, 82

Modified fast retransmit is a modification on TCP in the Linux kernel, suggested and implemented in *Petlund* [2009, a4] and *Evensen* [2008, a5]. It was introduced in the Linux kernel version 2.6.34 (*Petlund* [2010b, b2]) as `tcp_thin_dupack`. xviii, xxvi, xxx, 30, 32

multiplicative-decrease is a phase where the send rate is reduced drastically. 17, 21

Nagle's Algorithm 22, 23, 53, 57, 86, 88, 96, 113

network limited streams are network streams which send rate is limited by the TCP CWND. xxii, 11, 12, 57, 90

offloading schemes describes the various mechanisms used in networking for offloading certain tasks onto dedicated hardware. 88

- **checksum offloading** is the offloading of a checksum calculations onto dedicated hardware such as a NIC. 88, 89
- **Generic Receive Offload** is a generic implementation (not even restricted to TCP/IP) for merging incoming packets into larger segments. xxix, 93
- **Generic Segmentation Offload** is LSO implemented for protocols other than TCP. xxx, 93
- **Large Receive Offload** is a technique for reducing CPU overhead by having the NIC merge the data from multiple packets into larger segments before passing them on to the kernel. xxx, 93
- **Large Segment Offload** is a technique for offloading the segmentation of larger buffers onto the hardware of the NIC. xxx
- **TCP Offload Engine** is the offloading mechanisms implemented in NIC where parts of the processing of the TCP packets is moved off the CPU and into the integrated circuits of the NIC. xxxi
- **TCP Segmentation Offload** is LSO implemented for TCP. xxxii, 88, 93

One-Way Delay The time a packet uses through a network from one host to another. xxx

pfifo is a packet based queue where the first packet that comes in is the first packet to come out. 94, 105, 115, 127

queuing delay is the extra delay caused by the data spending time in a queue. xi, 19–21, 23, 28, 46, 47, 57, 86, 109, 112, 116–118, 122, 135

RDB prototype version 1 is the first RDB prototype implemented in Linux kernel version 2.6.22.1, and later ported to version 2.6.23.8 (*Evensen* [2008, a5]). The patch is included in code listing C.1. xviii, xxxi, 3

RDB prototype version 2 is the second implementation of RDB which we present in this thesis. xviii, xxxi, 3

RDBv2-CC is the CC implemented as part of RDBv2 presented in this thesis. 59, 76, 78, 82, 83, 87, 90

RTO Restart is a mechanism proposed for TCP and SCTP, aimed at providing fast loss recovery for connections with small amounts of outstanding data. xxxi, 30

RTO timer is the timer value for the RTO clock. xxii, 13, 17, 21, 23, 26, 27, 30–32, 35, 36, 94

send buffering delay is the extra delay caused by data segments being buffered in the senders TCP output queue. 86, 87, 119, 127

Skype is a popular voice-over-IP service. 8

slow-start is an algorithm which is used to control the growth of TCP's CWND. xi, 17, 18, 31, 33, 56

Smoothed RTT is an estimation of the RTT calculated by continually adjusting the estimated RTT value based on new RTTMs. This gives a stable RTT value that fluctuates in a smooth manner. xxxi, 23

ssthresh The slow start threshold for the CWND. When the CWND is greater or equal to this value, congestion avoidance is used. 17, 82

Tail loss probe is a TCP mechanism introduced in the Linux kernel version 3.10 *Dukkipati* [2013, b3]. xxxi, 30

TCP delayed acknowledgment is a mechanism in TCP where the ACK packets are delayed for a certain amount of time unless they can be piggy-backed on data packets. xi, 23–26, 35, 62, 63, 72, 113

TCP engine is the part of the Linux kernel that handles the data transferred by TCP. xi, xii, xviii, 3, 16, 39–41, 45, 52, 59, 61, 65, 77–79, 81–83, 90, 127, 133–135, 274

TCP friendly describes how a network flow that behaves similarly, in terms of aggressiveness, to a TCP stream in the same network conditions. 21, 29, 59, 69, 83, 85, 87, 104, 132

TCP output engine is the part of the Linux kernel that handles the data to be sent with TCP. See 2.7.1. xii, 40, 57, 66, 68, 88

TCP output queue is a linked list of SKBs that keeps track of the data that is either unsent, sent or un-ACKed. xxv, 3, 16, 24, 25, 33, 35, 36, 38–40, 42, 52, 53, 56, 57, 61, 65, 66, 68, 81, 86–90, 132, 276, 309

TCP Receive Window is the advertised receive window of one side of a TCP connection. xxxi, 15

TCP timestamps is an extension to TCP (*RFC7323*) that defines two header fields containing the sender timestamp and the *echo reply* timestamp which is used by the receiver side to indicate which incoming packet that resulted in the reply. xi, 24–26, 61–64

`tcp_early_retrans` is a TCP option in the Linux kernel This mechanism can be enabled through the sysctl variable `net.ipv4.tcp_early_retrans`. xxix, 30

`TCP_RTO_MIN` is the minimum RTO value for a TCP connection. 94

`tcp_thin_dupack` is a TCP option introduced in Linux kernel version 2.6.34 (*Petlund* [2010b, b2]). This mechanism is also referred to as Modified fast retransmit. It can be enabled through the socket option `TCP_THIN_DUPACK` or through the sysctl variable `net.ipv4.tcp_thin_dupack`. xxiv, 32, 59, 96

`tcp_thin_linear_timeouts` is a TCP option in the Linux kernel. This mechanism can be enabled through the socket option `TCP_THIN_LINEAR_TIMEOUTS` or through the sysctl variable `net.ipv4.tcp_thin_linear_timeouts`. See Linear Retransmission Timeout. xxiii, 59, 96

thin stream is a network stream that sends small amounts of data relative to greedy stream, such as produced when transferring files. These types of streams are application limited and often send segments smaller than one MSS. i, xvii, xxvii, 1–5, 12–14, 27–32, 36, 43, 45–47, 51, 53, 55–61, 68, 72, 73, 76, 83, 85, 87, 90, 91, 93–95, 97, 104, 105, 110, 115, 116, 118–120, 122–127, 132–135, 269

throughput is the total amount of data that is transferred from the sender to the receiver. The throughput may include the packet header size in addition to the payload for each packet. i, xxii, xxiii, 1, 4, 7, 9, 11, 19, 20, 28–30, 51, 53, 54, 69, 72, 73, 76, 78, 82, 83, 87, 90, 95, 104, 106, 118, 122, 124, 132, 134

Tools

`Analysetcp` is a tool for analyzing tcpdump trace files. 86, 99–101

`graph_r` is a collection of python scripts used for plotting the results. The source code is available at https://bitbucket.org/bendikro/graph_r. 95

`netem` is a network emulation tool in Linux that can be used to introduce arti-

ficial delay and loss in a network. xvii, 4, 46, 73, 91, 92, 104, 109, 115

ns-2 is version 2 in the network simulator series of discrete-event network simulators. 91

ns-3 is version 3 in the network simulator series of discrete-event network simulators. 91

R is a programming language for statistical computing. xxvii, 95

rpy2 is a python wrapper around the programming language R. 95

sshscheduler is a python script for running network tests. The source code is available at https://github.com/bendikro/sshscheduler 95

Streamzero is a client and server program developed for the purpose of testing thin streams. 73, 95, 99

tcpdump is a tool for saving traffic information in a network node. xxvii, 85, 86, 99, 105, 106

tcpprobe is a Linux kernel module written by Stephen Hemminger to gather different properties of a TCP stream at runtime. xxvii, 103

tcpproberdb is a Linux kernel module based on tcpprobe. 103

tcptrace is a tool for analyzing pcap traces produced by tcpdump. 100

wireshark is a tool for analyzing pcap traces produced by tcpdump. 100

Unreal Tournament 2003 is a FPS-game 10

Warcraft III is a game created by Blizzard Entertainment released in 2002. 10

Acronyms

ACK acknowledgment xviii, xxi, xxii, xxv, xxvi, xxix, 4, 13, 15–17, 21, 23–27, 31, 35, 36, 39, 40, 46, 52, 61–64, 72, 78, 81–83, 85, 86, 100, 103, 113, 133, 135, 274, 275, *Glossary:* acknowledgment

AIMD Additive Increase Multiplicative Decrease xxi, 17, 18, 29, 30, 57, 83

AQM Active queue management 52

BDP bandwidth-delay product 19, 91, 93

CAGR compound annual growth rate 8

CC congestion control xxi–xxiii, xxv, xxix, 1, 2, 4, 7, 8, 16, 17, 19–23, 27–30, 39, 43, 56–59, 69, 73, 76–78, 82, 83, 90, 99, 127, 128, 133, 134, *Glossary:* congestion control

CWND Congestion Window xi, xvii, xxi, xxii, xxiv, xxv, xxix, 17–21, 29, 30, 35, 52, 56, 57, 69, 71–73, 76, 78, 81–83, 86, 87, 95, 103, 105, 118–120, 128, 133, 265–267, *Glossary:* Congestion Window

DCCP Datagram Congestion Control Protocol 7, 8

DPIFL dynamic packet in flight limit xi, xvii, 60, 67, 68, 76, 83, 109, 115, 123, 127, 269

DSACK Duplicate Selective Acknowledgments xi, 16, 61, 63–65, 90, 135

dupACK Duplicate Acknowledgment xxii, 15, 17, 18, 25, 30, 32–35, 51, 52, 55, 61–64, 100, 106

ECN Explicit Congestion Notification 28, 52, 69

ER Early retransmit 30, 35, 45, 49, 51, 57, 133, *Glossary:* `tcp_early_retrans`

FACK Forward acknowledgment 16, 33

FPS-game first person shooter game xxvii, xxix, 10, 13, *Glossary:* first person shooter game

GRO Generic Receive Offload xxix, 93, *Glossary:* Generic Receive Offload

GSO Generic Segmentation Offload xxx, 93, *Glossary:* Generic Segmentation Offload

HOL blocking Head-of-line blocking xxx, 1, 2, 30, 33, 47, 51, 57, 104, 109, 110, 112, 113, 132, *Glossary:* Head-of-line blocking

IAT inter-arrival time xxx, 32, *Glossary:* inter-arrival time

IPSec Internet Protocol Security xxx, 89, *Glossary:* Internet Protocol Security

ITT inter-transmission time i, xi, xxx, 12, 13, 46, 47, 53, 60, 61, 72, 73, 76, 83, 86, 87, 94, 95, 97, 99, 105, 110, 116, 118–120, 122–125, 127–129, 132, *Glossary:* inter-transmission time

ITU International Telecommunication Union 8

LFN long fat network 19

LRO Large Receive Offload xxx, 93, *Glossary:* Large Receive Offload

LSO Large Segment Offload xxiv, xxx, *Glossary:* Large Segment Offload

LT Linear Retransmission Timeout xviii, xxvi, xxx, 30, 31, 42, 45, 49, 51, 55, 57, 76, 99, 133, 298, *Glossary:* Linear Retransmission Timeout

mFR Modified fast retransmit xvii, xviii, xxvi, xxx, 30–33, 35, 42, 45, 49, 51, 55, 57, 76, 99, 133, 298, *Glossary:* Modified fast retransmit

MIMD Multiplicative Increase Multiplicative Decrease 17

MMORGP Massively multiplayer online role-playing game xxiii, xxx, 10, 37, *Glossary:* Massively multiplayer online role-playing game

MSS Maximum Segment Size xxvi, xxxi, 14, 19, 23, 36, 52, 53, 72, 88, 128, 133

MTU Maximum Transmission Unit 12

NAT Network Address Translation 2, 8

NIC Network interface card xxiv, 88, 89, 91, 92

OWD One-Way Delay xxx, *Glossary:* One-Way Delay

PIF packet in flight 13, 14, 30–33, 35, 42, 53–55, 57, 59, 60, 73, 76, 83, 90, 103, 109, 110, 118

RDB Redundant Data Bundling i, xi–xiii, xxv, 2–5, 30, 36–38, 41–43, 45, 51–55, 59–65, 67–69, 72, 76, 77, 79, 81–83, 85, 87, 89, 90, 94, 96, 99–101, 104–106, 109, 110, 112, 113, 116–126, 128, 129, 132–135

RDBv1 RDB prototype version 1 xi, xviii, xxxi, 3, 5, 38, 39, 41–43, 45, 51, 57, 59, 65, 69, 76, 82, 83, 99, 298, *Glossary:* RDB prototype version 1

RDBv2 RDB prototype version 2 xii, xvii, xviii, xxiii, xxv, xxxi, 3–5, 59, 65–67, 69, 73, 76–78, 80–83, 91, 99, 104, 132, 133, 135, 265, 269, 274–279, 281, *Glossary:* RDB prototype version 2

RED Random early detection 52

RTO retransmission timeout xxii, xxv, xxvi, 13, 17, 18, 26, 27, 31, 32, 34, 35, 69, 113

RTOR RTO Restart xxxi, 30, 35, *Glossary:* RTO Restart

RTP Real-time Transport Protocol 7–9

RTT round-trip time xi, xxii, xxiii, xxv, 10, 11, 13, 14, 19–21, 23, 24, 29, 31, 32, 46, 47, 51, 55, 59, 60, 70, 72, 73, 83, 92, 94, 105, 113, 133, 134

RTTM round-trip time measurement xxv, 20, 23–26, 82

RTTVAR Round-trip time variation 26

RWND TCP Receive Window xxxi, 15, 19, 35, *Glossary:* TCP Receive Window

SACK Selective Acknowledgments xi, 15, 16, 20, 33, 35, 61, 63, 64, 90

SCTP Stream Control Transmission Protocol xxv, 1, 2, 35

SG Scatter-Gather 88

SKB socket buffer xxvi, 3, 24, 25, 36, 38–42, 53, 61, 65–68, 78, 88, 123, 127, 309

SMSS The sender sides MSS 17

SPIFL static packets in flight limit xvii, 55, 59, 67, 68, 76, 83, 109, 110, 112, 269

SRTT Smoothed RTT xxiii, xxxi, 23, 24, 31, *Glossary:* Smoothed RTT

TCP Transmission Control Protocol i, xxi–xxvii, 1–5, 7, 8, 10, 14–17, 19–21, 23, 24, 26, 27, 29, 30, 35–37, 39, 40, 43, 51–57, 61, 62, 69, 70, 72, 73, 76, 82, 83, 85, 87–91, 95, 99, 105, 106, 109–111, 113, 116–126, 129, 132–135

TFRC TCP-Friendly Rate Control 20–22, 56, 69, 70, 72, 76, 81, 82, 118, 119, 125, 128, 133, 280, 281

TFRC-SP TCP-Friendly Rate Control: The Small-Packet (SP) Variant xii, xv, xvii, 72, 73, 75, 76, 78, 82, 83, 134, 273

TLP Tail loss probe xxxi, 30, 35, 45, 51, 57, 133, *Glossary:* Tail loss probe

TOE TCP Offload Engine xxxi, *Glossary:* TCP Offload Engine

TSO TCP Segmentation Offload xxxii, 88, 93, *Glossary:* TCP Segmentation Offload

UDP User Datagram Protocol i, 1–3, 7–10, 88, 95

VNC Virtual Network Computing 8

VoIP Voice over IP i, 8, 9, 12, 14, 20, 72

WOW World of Warcraft xxxii, 10, *Glossary:* World of Warcraft

Chapter 1

Introduction

Today's public Internet has had a tremendous growth, from the first ideas of a global network in the 1960s, to the birth of TCP/IP in the 1980s, eventually leading to an explosion in the number of users starting in the mid-1990s continuing till today (*Internetworldstats.com* [2014, b4]).

Transmission Control Protocol **(TCP)** (*RFC793*), the most common protocol used on the Internet today (*John* and *Tafvelin* [2007, a6]), has mechanisms to control the send rate to prevent users from overflowing the network with too much data. V. Jacobson's work on *congestion control* **(CC)** for TCP in the late 1980s is, by many, recognized as a primary reason that enabled the Internet to grow at such a speed and size as it has (*Bhatti* et al. [2008, a7]; *Bansal* and *Balakrishnan* [2001, a8]).

The development of CCs and retransmission mechanisms for TCP has mainly focused on stability (fairness) and handling the transfers of the bigger and bigger amounts of data through the network, i.e., throughput (*Stewart* et al. [2011, a9]). This has left interactive applications that value latency over throughput in a bad spot. Many of the applications with strict latency requirements produce network traffic with thin-stream characteristics, meaning they send smaller and fewer packets compared to greedier streams. Due to the design of the mechanisms that are designed to prevent congestion collapse, such interactive applications suffer from higher latencies - we argue unnecessarily, or unfairly. In this thesis we focus on how to improve the performance for these types of applications.

1.1 Background and motivation

Alternative transport protocols to TCP and *User Datagram Protocol* **(UDP)** have emerged, that aim to replace them for certain uses. An example is *Stream Control Transmission Protocol* **(SCTP)**, which should be ideal for services that find UDP to be too basic, but do not require or want all of the functionality that TCP provides. With SCTP, ordering is optional, which eliminates the issue of *Head-of-line blocking* **(HOL blocking)** in TCP, which is beneficial for many types of interactive applications. With a variety of optionally negotiable features it could have the potential of replacing both UDP and TCP for many use cases.

A major problem is that many firewalls in home gateways and middleboxes, only support UDP and TCP, and do not let protocols such as SCTP through (*RFC3257*). Also, while OSes such as Linux, Solaris and FreeBSD and have SCTP included, Windows and OSX do not have native implementations. Until protocols such as SCTP are better supported in major OSes, home gateways, and middleboxes, they can not be used for services whose traffic must pass through firewalls and *Network Address Translations* **(NATs)**.

This gives a "Chicken or the egg" situation, where alternative protocols cannot gain wide adoption without better support by the network nodes. Meanwhile, manufacturers of operating systems and network middleboxes seem unwilling to invest resources into supporting new protocols until they are forced by the consumers.

A big challenge with deploying improvements to the Internet is the lack of any centralized control of all the nodes. As changes to nodes in the Internet must not break existing functionality, updates to protocols such as TCP must be backwards compatible. This lays heavy restrictions on what kind of changes that can be made to TCP.

Even with a clear trend showing an increase of audio and video streaming traffic, studies on the ratio of $\frac{UDP}{TCP}$ could not find a clear systematic trend showing a relative increase of UDP usage at the expense of TCP (*Lee*, *Carpenter*, and *Brownlee* [2010, a10]). This suggests that the common belief that UDP would be the obvious choice for streaming services might not be correct.

As TCP is still widely used, and is the *de facto* standard for many services that could benefit from the better latency provided by other protocols, we wish to look at how to improve the latency for such services using TCP.

Thin streams using TCP suffer from high latencies caused by the in-order guarantee that TCP provides. When packets are lost, the mechanisms for retransmitting the lost data cause considerable delays. Any data transmitted after the lost segment are subjected to HOL blocking which means that multiple data segments may be delayed due to only one lost packet.

In this thesis we present the continued work on the sender side TCP modification *Redundant Data Bundling* **(RDB)** for the Linux kernel. By enabling a more aggressive (re)transmission mode for thin streams, the per-packet latencies can be considerably improved. The modifications are made to maintain compatibility with TCP, which should allow for easy deployment into existing networks.

1.2 Problem statement

The Internet is a packet-switched network providing best-effort delivery of data packets. One of its strengths lies in how easily extendable the network is, and how robust the transfer of data is when one node in the network goes down. A weakness is that it heavily relies on the users to behave in a good manner. There is no centralized governance controlling how users behave, that can reprimand users that do not follow the "rules", partly because there really are no rules.

This is why controlling the traffic using an end-to-end CC is important. Due to the lack of any such mechanisms in UDP, trying to improve the performance for time-dependent traffic on protocols utilizing CCs is the best solution for the

network users as a whole. This will help the users that currently use TCP for such traffic, and reduce the incentives for application developers to choose UDP over TCP.

Mechanisms have been developed to improve the situation for interactive applications, one of them being RDB, which works by piggy-backing (bundling) already sent data in packets with new (unsent) data. The first RDB implementation, which we refer to as *RDB prototype version 1* **(RDBv1)**, showed great potential on improving the latency for thin streams, but it left certain issues unanswered. Uncertainties remain about the fairness of the RDB mechanism towards competing traffic. The lack of any mechanisms to limit aggressiveness, as well as the potential for abuse, is not sufficiently addressed in RDBv1, and is laid out as potential future work in the conclusion (*Evensen* [2008, a5]).

An implementation specific issue with *RDB prototype version 2* **(RDBv2)** also remains, regarding how the data contained in the *socket buffers* **(SKBs)** of the TCP output queue are modified by the mechanism. The operations required to perform the manipulations of the SKBs were deemed too intrusive in regards to data integrity.

Based on the earlier work on RDB, the goal of this thesis is to continue the study of improving the latency for thin stream traffic generated by interactive applications over TCP. With the previously mentioned issues of RDBv1 in mind, we specifically aim to:

- Develop an RDB implementation that is less intrusive than RDBv1, both in regards to data integrity of the buffers in the TCP output queue, and to the Linux kernel's TCP engine code. Streamlining the implementation, by better organizing the code into separated logical segments, is important to simplify the work of developing and extending the functionality as well as to make future patch submissions feasible.
- Investigate how to detect packet loss that is hidden from the current TCP implementation due to the redundant data introduced by RDB.
- Evaluate mechanisms that limit redundant data bundling to situations where it is most needed. This is to balance the aggressiveness of RDB against latency gains.
- Ensure that streams utilizing RDB are TCP-fair.

1.3 Research Method

The work in this thesis follows the *design* paradigm described in *Computing As a Discipline* by the ACM Task Force (*Comer* et al. [1989, a11]). This entails the process of stating the requirements (1) and specification (2) for the system we intend to create, before we design and implement the system (3), followed by evaluating (4).

We have written a prototype implementation of RDB in the Linux kernel, referred to as RDBv2, and experimentally evaluated the mechanism in a lab testbed, with a focus on the problems addressed in this thesis. We then analyse and present

the results based on traffic traces and run time information from the hosts on a multitude of different test setups.

Experiments

The experiments are performed in a testbed consisting of hosts running Debian Linux. We have set up the experiments with different configurations of sender hosts to test how the RDBv2 mechanism works in different scenarios.

We have run a set of latency tests with uniform loss rate enforced by `netem` to isolate the changes to the latency results by avoiding any external influence by competing network streams. The next set of experiments are set up with competing greedy and thin stream traffic to create a more realistic network environment, as well as to see how the RDB mechanism affects the other network streams. The last set of experiments, which we call fairness experiments, are designed to test how network streams produced by RDBv2 behave towards competing streams in respect to fairness when potential "evil-doers" try to abuse the mechanism to (unfairly) gain advantages over other competing network streams.

Data analysis

We have analysed the problem area in order to identify suitable metrics by which to evaluate the mechanisms. Then we have analysed the results of the experiments using these metrics. We calculate the ACK latency[1], the per-packet latency for the TCP streams, and compare the results of the different network streams. By calculating the goodput and throughput from the packet traces, we can compare the amount of data that the competing streams transfer through the network. We use this to try to identify unfair behavior of the RDB mechanism.

1.4 Main Contributions

The main contributions of this thesis can be summarized as follows:

- Implemented a TCP *congestion control* **(CC)** module for the Linux kernel, that enables a sender host to send redundant data in data packets already scheduled for transmission.
- Evaluated the thin-stream classification method in the Linux kernel and suggested improvements.
- Experimentally evaluated the RDB implementation with regards to:
  - The tradeoff between latency and aggressiveness in terms of increasing the packet size, and hence the throughout.
  - How it gives RDB streams significantly better latency without being unfair towards competing traffic.

[1] ACK latency is the time between a data segment is first sent onto the network, until an ACK for the data segment is received.

- How it prevents the RDB mechanism from being abused to obtain an advantage over competing traffic.

1.5 Outline

The thesis is structured as follows: In chapter 2 we look at the background for the different interactive applications that produce thin streams, and the mechanisms for improving the latency for such streams in the Linux kernel. We also present RDBv1, the TCP modification that our work is based on.

In chapter 3 we go into detail about what causes the increased latencies for thin streams, and aspects of the RDBv1 that we will try to improve upon. In chapter 4 we present RDBv2, the re-implementation of RDB, and in chapter 5 we evaluate the modifications by running experiments and presenting the results. Chapter 6 concludes the thesis with a summary of our findings and an outline on topics for future research.

Chapter 2

Thin streams

TCP is the most widely used protocol on the internet, and is the underlying engine for most common tasks such as, web browsing (HTTP), file transfers (FTP) and email (SMTP) (*John* and *Tafvelin* [2007, a6]).

The main focus of TCP's development has been on maximizing the throughput, i.e., move as much data as possible through the network as fast as possible. What matters on the receiving side when transferring data, such as a picture, is that the data is received correctly, and the time it takes to transfer the entire image, from the first packet is received, till the last packet is received. This is where TCP shines.

Time-dependent applications on the other hand, which usually send smaller segments of data, may depend on a minimal latency for each transferred data segment. For such use cases, TCP shines somewhat less.

2.1 Transport layer protocols for interactive applications

The conventional wisdom is to avoid TCP for interactive real-time applications. This is due to the overhead of TCP's reliability guarantee causing too much end-to-end delays (*Brosh* et al. [2010, a12]), which degrades the perceived quality of service as we see in section 2.2. The right choice of transport protocol highly depends on the type of application and its latency requirements, as well as the type of traffic it generates.

UDP is often used as an alternative to TCP, which gives better control over the network traffic. Because UDP provides a best-effort service with limited guarantees (only error checking), it is necessary to manually implement the required functionality. Speed, and better control over the network traffic, can be gained by using UDP, but it is costly to implement the extra functionality, and to do thorough testing to make sure it works correctly.

As UDP does not offer CC, it is not well suited for greedy stream transfers of large amounts of data, unless some CC mechanism is used on top, such as *Datagram Congestion Control Protocol* **(DCCP)**.

There are alternatives available that build upon UDP, such as *Real-time Transport Protocol* **(RTP)**, which is designed for delivery of data, over IP networks, with

real-time characteristics like interactive audio and video *RFC3550*. It provides features similar to TCP, such as timestamps, and sequence numbers to identify lost or out-of-order segments. A CC is also defined for RTP based on DCCP *RFC5762*.

However, any UDP solution still suffers from the problem of passing through firewalls and networks behind NAT, which is the reason why TCP often is used as a fallback when UDP can not be used (*Brosh* et al. [2010, a12]; *Guo* et al. [2006, a13])

2.2 Interactive applications with strict latency requirements

Many interactive multimedia applications utilize computer networks, where the user experience depends greatly on the network latency. Examples are *Voice over IP* **(VoIP)** software like Skype and Ventrilo, remote desktop control like *Virtual Network Computing* **(VNC)**, and online multiplayer games. All of these are examples of time critical services where the quality is highly dependent on timely arrival of network packets.

According to *Cisco* [2014, b5] forecasts, online gaming will have a *compound annual growth rate* **(CAGR)** of 34% from 2013 till 2018. They also forecast that internet video such as YouTube (short videos), Hulu (longer videos such as tv series) and Netflix (streaming), will have a CAGR of 29%, and by 2018, grown to more than $\frac{3}{4}$ of all internet consumer traffic. From 1993 till 2013, the international VoIP transported by VoIP carriers has had a CAGR of 13% (*TeleGeography Report* [2013, b6]).

2.2.1 VoIP

Based on research using the E-model, the *International Telecommunication Union* **(ITU)** recommends an upper limit of 400 ms end-to-end (one-way) delay for conventional speech (*(ITU-T)* [2003, b7]). Figure 2.1 shows that the quality begins to degrade at around 200 ms one-way delay. At about 300 ms one-way delay some users are getting dissatisfied and at 400 ms, many are getting dissatisfied. At 500 ms nearly all the users are dissatisfied. The ITU states that most interactive applications will not be affected negatively if the end-to-end delay is below 150 ms. However, because of the lack of standardized or agreed-upon assessment tools for non-speech applications, the effects of latency cannot be stated as clearly.

Video-conferencing, such as Skype, which is simply an extension of VoIP that includes video as well as audio, has the same latency requirements to VoIP. In addition to the requirements for audio quality, users will also evaluate the user experience based on the video quality, and how well it matches the audio, i.e., how synchronized they are.

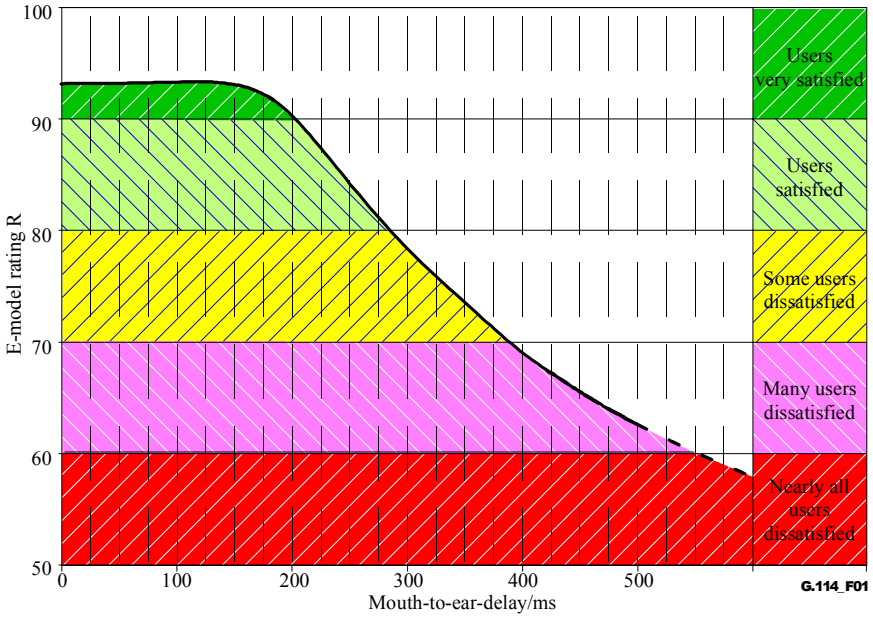


Figure 2.1: G.114 – Determination of the effects of absolute delay by the E-model (*(ITU-T)* [2003, b7])

2.2.2 Video streaming

Video streaming is a service that has grown tremendously the last years, pushed by providers such as YouTube, Netflix and HBO. Cisco forecasts that by 2018 over half of all internet video traffic will be content delivery network traffic (*Cisco* [2014, b5]). While streaming content such as movies and series does not have the same strict latency requirements as VoIP, video streaming is still very vulnerable to variations in the throughput. Video that stops playing to wait for the next video frames to arrive, is highly disruptive for the user experience. Such *stalling* issues are dealt with by buffering parts of the video by requesting a given length of the video in advance. *Dobrian* et al. [2011, a14] find that the importance of different quality metrics, like time spent buffering, buffering event rate and rendering rate (frames per second), depend on the type of content that is streamed, like short or long video and if it is streamed live.

While one may argue that on-demand video streaming does not have strict latency requirements, as the video can simply be buffered and played with a delay. If the viewers do not mind the delay, this is true, but in situations such as live soccer matches, a delay may reduce the viewing experience to a great deal. Imagine viewing the live video with a 10 seconds delay and having to listen to the neighbors' reactions on important game events before you can see it yourself.

A common belief or expectation has been that streaming traffic would use UDP, or protocols that rely on UDP, such as RTP. In light of the increase in audio and video streaming traffic in the later years, *Lee*, *Carpenter*, and *Brownlee* [2010, a10] studied the ratio of $\frac{UDP}{TCP}$, but could not find a clear systematic trend showing

a relative increase of UDP usage at the expense of TCP.

Despite the shortcomings of TCP it is still widely used for video streaming, where reports suggest that already in 2006, as much as 50% of video streaming on the Internet was carried by TCP (*Guo* et al. [2006, a13]). *Cisco* [2014, b5] forecasts, that by 2018, 79% of the global consumer internet traffic will be IP video traffic.

2.2.3 Online games

There are a multitude of different online games where the gameplay quality depends on the network in a varying degree. When the game traffic is delayed too much, the players experience what is called lag.

M. Claypool and *K. Claypool* [2005, a15] categorizes online games into three categories based on how dependent they are on the latency of the transferred data:

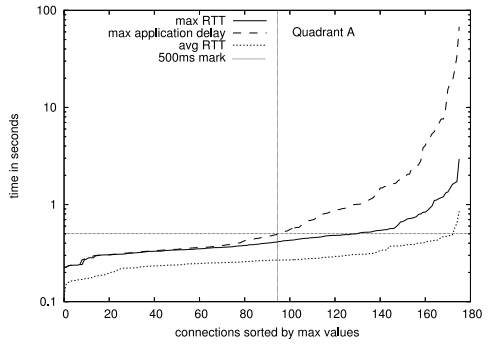
- ***First person shooter games*** **(FPS-games)** like Half-Life and Quake where the player navigates a virtual world in a first person perspective.
- **Role-playing games**, like *World of Warcraft* **(WOW)**, Age of Conan and Anarchy Online, where the player usually controls an avatar in third person view.
- **Real-time strategy games** and construction games, like Warcraft III, with an omnipresent control.

They find that for first person shooter games, 100 ms one-way delay is a maximum threshold for a good user experience. For role-playing games like WOW and Anarchy Online, the threshold is around 500 ms, and for real-time strategy games it is around 1000 ms.

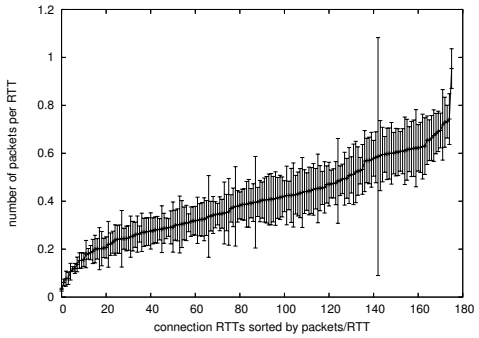
The *first person shooter games* **(FPS-games)** have very strict latency requirements, as the actions the players perform, like aiming and shooting a moving target, must be completed within a strict time frame. *Quax* et al. [2004, a16] test the effects of latency on the gameplay quality in the FPS-game Unreal Tournament 2003., and find that from 60 ms *round-trip time* **(RTT)** and on, the players experience the delay as disturbing.

Massively multiplayer online role-playing games **(MMORGPs)** can have thousands of players interacting at the same time, requiring continuous updates of game data. The information sent from the player is everything the user does, like starting to move, which direction the player moves, when the player stops, if the player shoots and the direction of the shot. Position info for the player must be sent to the server, and position info for other players must be sent back. All this information must be sent quickly for the game play to be smooth, and only small delays or loss of this information can be noticed by the players.

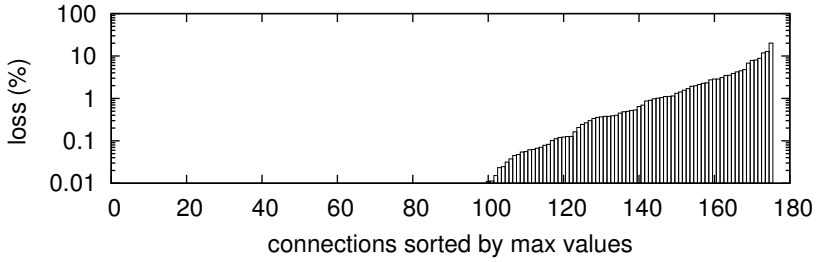
Griwodz and *Halvorsen* [2006, a1] investigate the data traffic of the game Anarchy Online. They analysed a one-hour server-side trace from a game server located in the US, that contains 175 TCP connections. In figure 2.2 we see the results of the analysis. Subfigure 2.2.(a) shows that quite a lot of the connections



(a) RTT versus maximum application delay.



(b) Packets per RTT with standard deviation.



(c) Per-stream loss rate.

Figure 2.2: Statistics from analysis of Anarchy Online server side dump (*Griwodz* and *Halvorsen* [2006, a1]; *Petlund* [2009, a4])

experience latencies above 500 ms (quadrant A), the threshold for a good user experience. Subfigure 2.2.(b) shows the number of packets per RTT the connections have.

2.3 Characterizing different network streams

Applications performing bulk data transfer such as FTP, where the primary goal is to transfer as much data in as little time as possible, produce what is called greedy streams. By continually pushing the network to transfer the data as fast as possible, the network becomes the limiting factor, hence we call them network limited streams.

Applications that have a finite bandwidth requirements, produce network streams that are application limited, meaning the throughput is limited by how much data the application produces, and not by how much data the network is able to transfer. *RFC2861* defines the term *application limited* period as "when the sender sends less than is allowed by the congestion or receiver windows" and *network limited period* as "any period when the sender is sending a full window of data.".

One may argue that any stream that is not greedy, i.e., not trying to achieve the maximum possible throughput, is application limited, making it an inherent

Table 2.1

Application	Protocol	Payload size (bytes)			Packet inter-arrival time (ms)						Avg bandwidth requirement	
									Percentiles			
		avg	min	max	avg	med	min	max	1	99	(pps)	(bps)
Casa (sensor network)	TCP	175	93	572	7287	307	305	29898	305	29898	0.137	269
Windows Remote Desktop	TCP	111	8	1417	318	159	1	12254	2	3892	3.145	4497
VNC (from client)	TCP	8	1	106	34	8	0	5451	0	517	29.412	17K
VNC (from server)	TCP	827	2	1448	38	0	0	3557	0	571	26.316	187K
Skype (2 users)	UDP	111	11	316	30	24	0	20015	18	44	33.333	37K
Skype (2 users)	TCP	236	14	1267	34	40	0	1671	4	80	29.412	69K
SSH text session	TCP	48	16	752	323	159	0	76610	32	3616	3.096	2825
Anarchy Online	TCP	98	8	1333	632	449	7	17032	83	4195	1.582	2168
World of Warcraft	TCP	26	6	1228	314	133	0	14855	0	3785	3.185	2046
Age of Conan	TCP	80	5	1460	86	57	0	1375	24	386	11.628	12K
BZFlag	TCP	30	4	1448	24	0	0	530	0	151	41.667	31370
Halo 3-8 players	UDP	247	32	1264	36	33	0	1403	32	182	27.778	60223
Halo 3-6 players	UDP	270	32	280	67	66	32	716	64	69	14.925	35888
World in Conflict (from server)		365	4	1361	104	100	0	315	0	300	9.615	31K
World in Conflict (from client)		4	4	113	105	100	16	1022	44	299	9.524	4443
Test Drive Unlimited	UDP	80	34	104	40	33	0	298	0	158	25.000	22912
Tony Hawk's Project 8	UDP	90	32	576	308	163	0	4070	53	2332	3.247	5812

Table 2.1: Examples of thin stream packet statistics based on analysis of packet traces. (*Petlund* [2009, a4])

property of the application. With that definition, a stream may be both application limited and network limited.

With the increase in bandwidth capacity, both in the Internet and in consumer homes, more applications that were previously network limited, such as VoIP, are no longer so.

2.3.1 What is a thin stream?

Application limited streams is a very broad class of streams, which leads us to further classifying the types of streams in this category. We do this by looking at what we call the thickness of a stream, which is how much data is sent and how often.

Applications with strict latency requirements, have in common that they often produce application limited traffic with thin-stream characteristics. This is traffic consisting of small packets with a relatively high *inter-transmission time* **(ITT)**, as we see from the examples in table 2.1.

In table 2.1 we see payload size and packet ITT statistics for many different applications and games that send traffic with thin-stream characteristics. We see that typical characteristics of the packets is the small payload size, often between a few tens to a few hundred bytes in total which is well below the *Maximum Transmission Unit* **(MTU)** of 1500 bytes for IP datagrams over Ethernet (*RFC894*). We also see that the ITT of the different applications is varying to a great degree, from

an average of 20-30 ms at the lowest, to many hundreds and even thousands at the highest.

Lang, *Branch*, and *Armitage* [2004, a17] analyse the traffic for the FPS-game Quake3. They find that the ITT of the traffic from the server to the clients is very regular. Independent of how many clients are connected the server sends a packet to each client every 50 ms. The IP datagram size would vary depending on the number of clients connected. For the data traffic from clients to the server, the ITT varies in the range of 10-60 ms depending on the computer hardware (graphics card), and the map the players were playing on. The IP datagram size is in the range 50-70 bytes and does not vary depending on the computer or number of players connected.

In the experiments presented in *Griwodz* and *Halvorsen* [2006, a1] on Anarchy Online, they find that most of the retransmissions are caused by *retransmission timeouts* **(RTOs)**, which suggests that most of the game traffic has few *packets in flight* **(PIFs)** since they are unable to trigger a fast retransmit.

2.3.2 Identifying thin streams

While identifying thin streams might seem like a manageable task, it is not so simple. Applications producing thin streams are very diverse, but so is the traffic they produce. *Fuchs* [2014, a18] study how to characterize, identify and classify thin streams. They present the following possible characteristics and metrics for identifying thin stream:

- **PIFs or flight size**
  The PIFs is the number of outstanding packets or packets in transit, i.e., packets that are in transit to the receiving host, or whose *acknowledgments* **(ACKs)** are in flight on its way back to the sender.
  In many cases, the PIFs are similar to the flight size, where flight size is the amount of outstanding data in the network (as defined in *RFC5681*). When doing segment based accounting, contrary to bytes based accounting, the flight size is the number of packets that have not been ACKed.
  However, in some cases, especially in thin-stream scenarios, they are not at all the same. For a stream with in ITT of 100 ms, where the RTT is always exactly 100 ms, and the RTO timer is 350 ms, the PIFs will most of the time be 1. As long as no packets are lost, the PIFs and flight size will both be 1. However, when a packet is lost, the flight size will increase to 2 when the sender sends a new packet exactly 100 ms after the previous packet was sent. The PIFs will still be 1. 100 ms later, after a new packet is sent, the flight size will increase to 3, but the PIFs is still 1.
- **Packet inter-transmission time**
  The ITT together with the RTT is for the most part what controls the PIFs. The ITT only tells you what the application needs to send within a specific time period. An application producing 100 bytes of data, with an ITT of 5 ms on a connection with 150 ms RTT, will at most have 30 PIFs. Compared to the applications in table 2.1, this stream would not be considered thin. If

however, the link has an RTT of 10 ms, the PIFs would be 2, which does not seem very high. A greedy stream on a good link will have have tens of PIFs at a minimum, but if we move that same application to a link with a very limited capacity, the PIFs could reach levels as low as streams we would consider thin.

- **Payload and packet size that often is below the *Maximum Segment Size* (MSS)**
  As all greedy streams will fill up each packet with the MSS of data, this is possibly one of the best indicators for thin streams.
  However, there are still scenarios where it does not work well. For example, an application may produce relatively small amounts of data (compared to greedy streams) in a bursty fashion, such as 3 * MSS, once every RTT. In total it would not be sending much data, and with most of the time not sending any data it might be considered thin, even when most packets are full MSSs.
- **Stream duration**
  For interactive applications such as games or VoIP, one can expect a certain duration of the network stream. Using the duration requires a preliminary goal for which types of applications you want to include in the class. HTTP transactions for example, often have a very short duration, but they do not transfer much data in total. *Dukkipati*, *Mathis*, et al. [2011, a19] find that the average HTTP response from Google's servers was 7.5kB, which corresponds to about 5-6 TCP segments. While few would argue that latency is not important for HTTP transactions, it is special case in terms of thin-stream classification, at least in comparison to the applications in table 2.1.

2.4 Overview of TCP

TCP provides important services that the underlying IP, in its unreliable nature, does not. In short, these features are:

- **Connection oriented**
  The hosts set up a connection before sending data. They may also negotiate and agree on different parameters during the connection establishment.
- **Stream oriented**
  The application sends and receives a stream of data and does not need to know how the data is transferred by the underlying network layers. The data is delivered to the application layer in the same order was it was sent.
- **Reliable transmission**
  In case of packet loss or packet reordering, the receiving side waits until any gaps in the order av filled before delivering the data to the application layer. The transmitted data is verified by checksums to guarantee that the data is correct. If the sender notices loss it will retransmit the data.

- **Flow control**
  The receiver side notifies the sender of how much data it will accept by setting the *TCP Receive Window* **(RWND)** field in the TCP header.

- **Congestion control and avoidance**
  The senders way of controlling the send rate to avoid overflowing the network with more traffic than can be handled. This is arguably the most critical and complex part of TCP in regards to how well the network will work when shared by multiple network streams.

2.4.1 Data flow

To meet the requirement for reliable transmission, the TCP header in each packet contains a field for the sequence number of the payload and the length. This is used to keep track of what parts of the data stream has been sent and what has been verified as received by the receiving end.

The header has a acknowledgment field used to tell the sender what has been successfully received. This cumulative acknowledgment tells the sender that all the data up to the sequence number in the ACK field has bee received successfully. If a packet is lost, packets will arrive out-of-order on the receiver side, which means that there is a gap in the sequence space of the data. The receiver will keep this data in the incoming buffers and immediately reply with an ACK packet where the acknowledgment field contains the sequence number of the first byte starting the gap of missing data in the sequence space. When the sender host receives an ACK where the acknowledgment field contains a sequence number that has already been ACKed by a previous ACK packet, it counts it as a *Duplicate Acknowledgment* **(dupACK)**.

When a data segment that fills a hole in the sequence space arrives at the receiver side, it will immediately reply with an ACK packet where the acknowledgment field contains the sequence number of the first byte that has not been received in-order, i.e., the next expected byte.

SACK

Selective Acknowledgments **(SACK)** is an extension to TCP, which allows the receiver side to provide more detailed information to the data sender, about which data segments have been received (*RFC2018*). By using the optional parts of the TCP header, the data receiver can provide a set of sequence number pairs defining one or more data segments that have arrived out-of-order.

When a packet is lost, creating a gap in the sequence space of received data, the (cumulative) acknowledgment field of any packet sent in return must contain the last byte received in-order + 1. This applies even when newer data is received. *Selective Acknowledgments* **(SACK)** does not change this, but with by providing pairs of sequence numbers in the SACK header option, the sender will also know which segments have been received, and hence, which do not need a retransmission. While the SACK option is only advisory, and permits the data receiver to

discard the data blocks that cannot be cumulatively ACKed, doing so is discouraged unless the receiver is forced to, such as when buffer space is limited. Because the SACK option is advisory, the data sender may not discard any data in the TCP output queue that was reported in a SACK-block. Only when data is ACKed by the cumulative acknowledgment field, may it be removed.

SACK is widely used on the Internet, where analysis of the traffic on Google's Web servers, found that 96% of the connections negotiated SACK (*Dukkipati*, *Mathis*, et al. [2011, a19]).

DSACK

RFC2883 defines an extension to SACK that specifies *Duplicate Selective Acknowledgments* **(DSACK)**-blocks which are SACK-blocks with sequence numbers that are lower than the cumulative sequence number of the acknowledgment field. This enables the data receiver to report about duplicate packets or data segments to the sender. In the Linux kernel, the TCP option `tcp_dsack` is enabled by default.

FACK

Forward acknowledgment **(FACK)** aims at improving the performance of TCP Reno, when multiple packets are lost. FACK relies on SACK information to keep track of which blocks are held by the receiver. In addition, it separately accounts for the amount of outstanding retransmitted data. By considering any gaps in the reported SACK blocks as an indication of loss, FACK's packet accounting depends on in-order delivery. Due to this, the Linux kernel disables FACK when reordering is detected.

While *Mathis* and *Mahdavi* [1996, a20] present FACK as a CC, the Linux kernel does not implements the FACK functionality as a separate CC, but as a part of the core TCP engine. Through the sysctl option `net.ipv4.tcp_fack`, FACK may be disabled.

2.4.2 Congestion Control

The CC mechanism in TCP, is the mechanism responsible for keeping the data flow in control to avoid overflowing the network, and in turn causing congestion. The principles of the CC is based on controlling the network resources, with the goal of an optimal fair allocation for the network as a whole. The first CC mechanisms for TCP were developed out of necessity, after the congestion collapse that hit the Internet in the mid 1980s (*Jacobson* [1988, a21]). The stability of today's Internet is often attributed to the CC and congestion avoidance mechanisms made famous by Van Jacobson (*Bhatti* et al. [2008, a7]; *Bansal* and *Balakrishnan* [2001, a8]; *Mcdonald* and *Nelson* [2006, a3]), along with further improvements in modern versions (*Floyd* and *Fall* [1999, a22]).

2.4.2.1 Basis for the modern congestion control in TCP

The basis for the modern CC mechanisms in TCP consists of the following (as described in *RFC5681*)

- **Slow-start**
  The first phase of a TCP stream's development is slow-start. This phase follows the *Multiplicative Increase Multiplicative Decrease* **(MIMD)** algorithm where the stream will build the *Congestion Window* **(CWND)** exponentially until it reaches the network limit. It keeps building the CWND until it experiences loss or the CWND has reached the ssthresh.

- **Congestion avoidance**
  After the initial slow-start phase, the congestion avoidance phase controls the CWND. This mechanism is used as long as the CWND is greater than ssthresh.

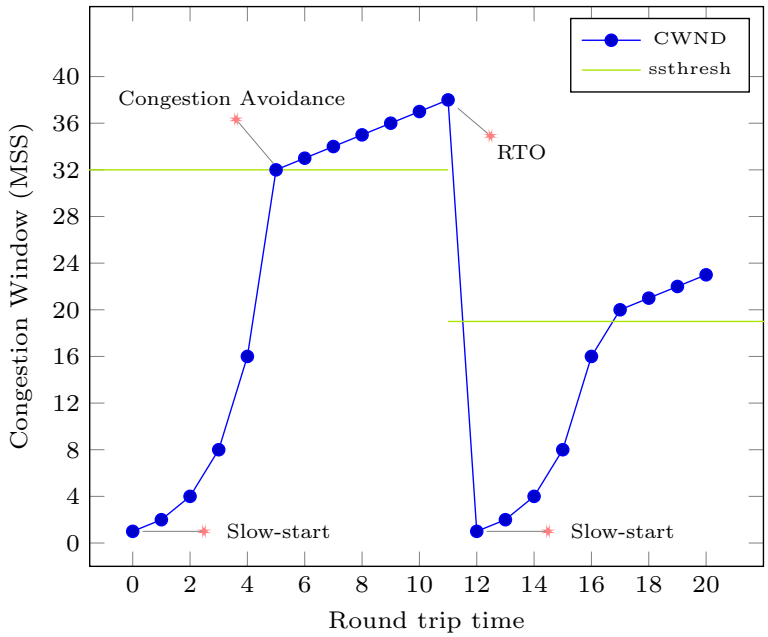
  This phase follows the *Additive Increase Multiplicative Decrease* **(AIMD)** algorithm described in *Jacobson* [1988, a21] where the CWND is grown at a slow rate (additive-increase) and reduced to half the send window size (multiplicative-decrease) if case loss occurs. When an RTO is triggered, the CWND is adjusted as shown in subfigure 2.3.(a).

- **Fast retransmit**
  In the case of packet loss, the sender must retransmit the data at some point. To avoid having to wait for the RTO timer to trigger, the fast retransmit algorithm will use *Duplicate Acknowledgments* **(dupACKs)** to detect loss. If a packet is lost, the receiver notices a gap in the sequence numbers of the received packets. For each subsequent packet received that is not in-sequence on that connection, the receiver replies with a dupACK, until the missing segment is filled. With fast retransmit, a retransmission is triggered when three consecutive dupACKs are received. To avoid triggering a retransmission in case of packet duplication, or when packets arrive out-of-order (caused by different network routes), the limit is set to three dupACKs (*RFC5681*).

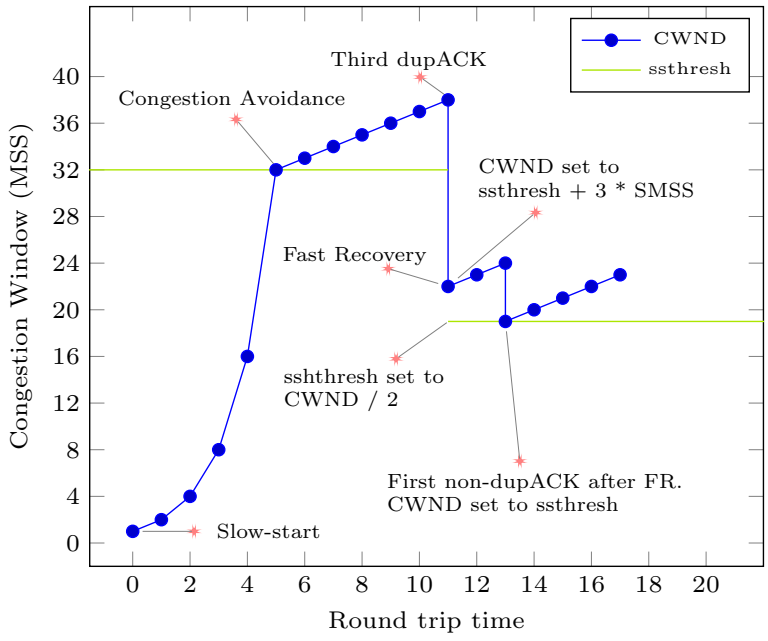
- **Fast recovery**
  The goal of this mechanism, is to allow the sender to back off in a more gentle way when sporadic loss occurs. Instead of going to slow-start, as is done after an RTO, the sender may continue to transmit at a lower rate, as long as packets are coming through the network.

  Immediately after a retransmission is initiated by fast retransmit, the ssthresh must be set to no more than what is given in equation 2.1, and the CWND set to ssthresh plus 3 * SMSS. When the first ACK that acknowledges new data is received, the CWND is set to ssthresh. Given that the CWND is set to ssthresh when new data is ACKed, congestion avoidance is used to increase the CWND further on.

  Subfigure 2.3.(b) shows an example of how fast recovery works.



(a) RTO triggers a CWND reduction



(b) Three dupacks triggers fast retransmit and fast recovery

Figure 2.3: Illustrations of Slow-start, and how an RTO, and three dupACKs are handled in the AIMD phase.

$$ssthresh = max(FlightSize/2,\ 2 * SMSS) \tag{2.1}$$

2.4.2.2 Congestion Window

The CWND is used to control how much data a host can send, by specifying the upper limit of outstanding data the sender may have. The measurement unit for the CWND can be either segments or bytes, and, while the *RFC5681* specification is for the most part general in respect to the units for the CWND, some equations are defined with bytes in mind. In the Linux kernel, the CWND is calculated in segments of size MSS, which is the unit we will use in this thesis unless otherwise specified.

The CWND, together with the advertised RWND, is what controls the send rate of a connection. By continuously calculating the TCP send window (sliding window) to be the minimum value of the CWND and the RWND, both the capacity of the network, and of the receiver side host, is taken into account when adjusting the send rate.

2.4.2.3 Different types of congestion control mechanisms

TCP Reno was the first CC algorithm to support all the four mechanisms described in section 2.4.2.1 *RFC2001*. TCP New Reno, which improves upon TCP Reno by better handling multiple loss within the same window, was the default CC in the Linux kernel until version 2.6.8 (Released August, 2004), when it was replaced by TCP Cubic. TCP Cubic uses a different algorithm to calculate the CWND growth making it better suited for networks with large *bandwidth-delay product* **(BDP)**, commonly known as *long fat network* **(LFN)** (*Ha*, *Rhee*, and *Xu* [2008, a23]). These traditional CC mechanisms are loss-based, meaning that they primarily rely on loss detection for estimating the amount of congestion in the network. An inherent problem with the loss based CC, is that they must cause loss to find the limitations of the network. This also causes loss for other network streams using the same resources.

Research has been done on trying to find alternative CCs for TCP, that rely on other metrics than loss, or loss alone, to measure congestion. One example is TCP Westwood, which aims for a better handling of sporadic and random loss. By avoiding the drastic reduction of the CWND that TCP New Reno does, it takes into account the estimated available bandwidth when adjusting the CWND. Tests have shown that TCP Westwood provides considerably better throughput on wireless networks (*Casetti* et al. [2002, a24]).

Delay based congestion control

TCP Vegas is a delay based CC, that uses the calculated queuing delay to adjust the CWND. TCP Vegas implements a more accurate RTT calculation compared to TCP Reno, which is required for the queuing delay calculations. It compares

the measured throughput with the expected throughput by using the *baseRTT*, the smallest *round-trip time measurement* **(RTTM)** within the last RTT, and the current CWND (which is assumed to be analogous to the bytes in transit). The calculated *diff* is the difference between the current CWND and the desired CWND, where the size of the *diff* indicates the amount of queuing delay or network congestion.

Variations on the CWND reduction

Alternative mechanisms, to control the CWND reduction in the fast recovery phase, have been created. *RFC3517* defines "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP", an algorithm that uses SACK information to increase performance by letting senders recover more effectively when losing multiple segments within a window. "Rate halving" is another mechanism, where the sender alternates between sending retransmissions and new data during the recovery RTT. *Dukkipati*, *Mathis*, et al. [2011, a19] proposes *Proportional Rate Reduction*, which improves upon the standard fast recovery in cases of bursty loss and losses at the end of short flows, where the application has no more data to send. *Proportional Rate Reduction* became the standard recovery mechanism in Linux kernel version 3.2.

2.4.2.4 Equation based Congestion Control

Historically, the development of end-to-end CC and TCP goes hand in hand, which has left other protocols in the shadows, so to speak. Nevertheless, the need of CCs for non-TCP streams has been emerging, ever since the use of latency-sensitive applications started to manifest itself as a major "player" on the Internet.

The need to define how a CC should behave in a formal way has lead to a new type of CCs. Equation based CCs is a class of CCs, where the send rate is adjusted based on a function on some property, such as loss even rate. TCP is known to constantly and aggressively probe the network to find the maximum send rate. The primary goal of the equation based CCs is to avoid this, by finding and maintaining a steady send rate, that does not fluctuate as much as TCP's, while still reacting to network congestion.

TCP-Friendly Rate Control

Floyd et al. [2000, a25] presents *TCP-Friendly Rate Control* **(TFRC)**, which is a CC for non-TCP streams operating in a best-effort network, such as the Internet. It is designed for applications that not only value maximum throughput, and it specifically aims to reduce the variations in throughput compared to TCP.

Streaming media and VoIP are such examples, where stable throughput is very important to maintain a smooth playback of video or audio. TFRC is designed to be "reasonably fair when competing for bandwidth with TCP flows," (*RFC5348*) where a reasonably fair stream is defined as "if its sending rate is generally within a factor of two of the sending rate of a TCP flow under the same conditions".

Most importantly, TFRC is designed to avoid the drastic reduction of the CWND of TCP's multiplicative-decrease phase, when loss is detected.

To achieve this, TFRC uses a control equation (equation 2.2) presented by *Padhye* et al. [1998, a26]), which is a TCP response function describing the steady-state sending rate of TCP.

$$T = \frac{s}{R\sqrt{\frac{2bp}{3}} + t_{RTO}(3\sqrt{\frac{3bp}{8}})p(1+32p^2)} \tag{2.2}$$

T is the upper send rate (bytes/s), s the packet payload size, R the RTT, p the loss event rate, b the maximum number packets acknowledged by a single TCP acknowledgment, and t_{RTO} the current RTO timer in seconds.

Instead of "blindly" adjusting the send rate (CWND) when loss occurs, TFRC uses equation 2.2 to calculate the send rate based on the rate of lost packets (loss event rate), the RTT, and the packet size. The receiver has the responsibility of calculating the loss event rate and report this to the sender regularly in feedback packets. The loss event rate is then used by the data sender to adjust the sending rate. The reason these calculations are made on the receiver, is that the loss calculations are not sensitive to lost ACK or feedback packets.

To reduce oscillation in the queuing delay, it is recommended to reduce the send rate when the queuing delay increases. This is done by maintaining an RTT estimation of the square root of the RTT, and comparing that to the square root of the most recent RTT sample.

2.4.2.5 Binomial Congestion Control Algorithms

Bansal and *Balakrishnan* [2001, a8] introduce a class of congestion control algorithms they call binomial algorithms. They generalize TCP's additive-increase and multiplicative-decrease using two properties k and l, where an algorithm is TCP-compatible (TCP friendly) when it satisfies $k+l=1$.

Additive-increase is generalized by increasing inversely proportional to the power k of the current window, and multiplicative-decrease, by decreasing proportional to the power k of the current window. This is formalized as

$$\begin{aligned} I &: \omega_{t+R} \leftarrow \omega_t + \alpha/\omega_t^k; \alpha > 0 \\ D &: \omega_{t+\delta_t} \leftarrow \omega_t - \beta\omega_t^l; 0 < \beta < 1 \end{aligned} \tag{2.3}$$

where I is the increase after receiving ACKs for a window within an RTT, D is the decrease of the window after a loss/congestion event. ω_t is the window size at time t, R is the RTT, and α and β are constants or functions of the current window

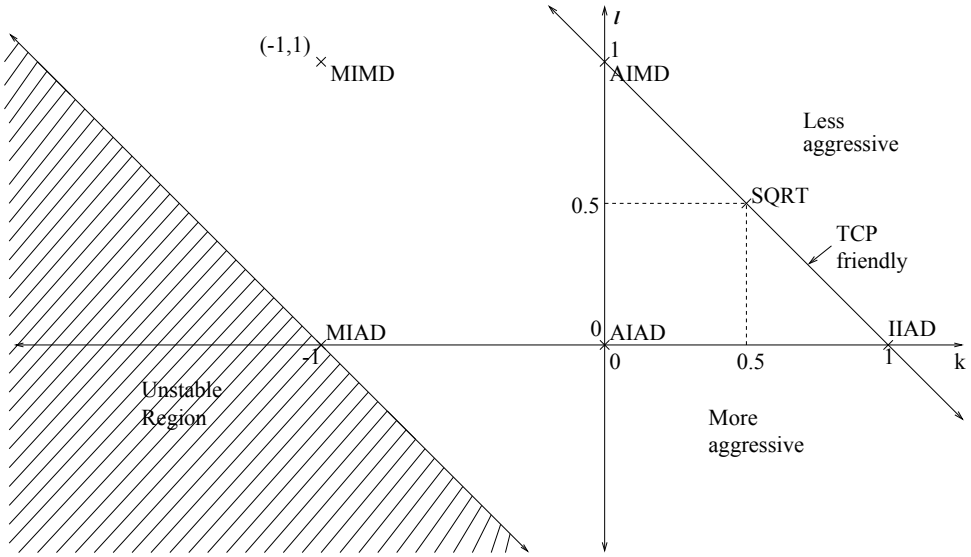


Figure 2.4: The k,l space of nonlinear controls from the binomial algorithms, with the $k+l=1$ line showing the set of TCP-compatible controls. (*Bansal* and *Balakrishnan* [2001, a8])

size. Equation 2.3 generalizes the linear control algorithms:

$k=0, \quad l=1$: AIMD (TCP)
$k=-1, \quad l=1$: MIMD (multiplicative increase/multiplicative decrease)
$k=-1, \quad l=0$: MIAD
$k=0, \quad l=0$: AIAD

Figure 2.4 shows *a)* the k,l space of nonlinear controls *b)* where it corresponds with the four linear algorithms, and *c)* the $k+l=1$ line which is the set of TCP–compatible controls.

The idea is that by adjusting the k and l properties, the CC can be made to better fit the needs of applications such as audio and video streaming where a drastic reduction of the send rate on loss events is very damaging.

An important difference with the binomial CC compared to other equation based CCs such as TFRC is that it does not make use of a calculated loss rate.

2.4.3 Nagle's algorithm

Nagle's Algorithm is a mechanism proposed by John Nagle, to address *The small-packet problem* described by *RFC896*. The problem, described as early as 1960s, is caused by the sender application doing multiple send calls to the kernel with very small segments. In a scenario, such as an SSH session, depending on the implementation, typing a single character could cause the application to do a send

system call. This would result in a TCP to be sent, where the header size (of around 40 bytes) is 40 times larger than the payload, giving a overhead of 4000%.

To reduce the number of transferred packets, Nagle's Algorithm causes the data from the application layer to be buffered in the kernel for a certain period, before being sent. As long as there are outstanding packets, Nagle's Algorithm will buffer data until a full-sized packet (defined by the MSS) can be sent.

The Minshall variation on Nagle's algorithm

The Linux kernel uses a variation of Nagle's Algorithm defined in *Minshall* [1999, c17]. This modification aims at fixing a problem that can occur when a sender using Nagle's Algorithm interact with a receiver using TCP delayed acknowledgment. If an application sends $1.5 * MSS$ worth of data to the kernel, the kernel will send one full-sized packet, and delay the rest of the data. Not until the kernel either receives more data from the application layer to fill a full packet, or the first packet is ACKed, will the rest of the data be transmitted. With TCP delayed acknowledgment enabled, the data receiver will delay the ACK until more packets arrive or the delay-limit is reached, which causes grave delays.

The modification to Nagle's Algorithm therefore proposes to buffer data only when there is an outstanding packet smaller than the MSS. In this case, all the $1.5 * MSS$ worth of data from the application will be sent immediately. This is because when testing for the second segment, there are no less-than MSS-sized packets in transit.

The variation to Nagle's Algorithm is enabled by default in the Linux kernel.

2.4.4 Delayed ACKs

TCP delayed acknowledgment is a mechanism designed to reduce the amount of packets in the network by sending fewer ACKs. Without TCP delayed acknowledgment, the receiver host will send an ACK-reply to the sender for each incoming data packet. By allowing the receiver to introduce a delay of up to 500 ms (*RFC1122*), it can reduce the amount of pure-ACKs (packets that have no payload data) by ACKing the data from multiple packets received within a certain time frame. If the receiver is also sending data packets (within the time frame of the threshold), it can avoid sending pure-ACKs by piggy-backing the ACK on the data packets.

2.4.5 RTT measurements

An important property for the CC algorithms is the RTT, which is used for calculating the RTO timer.

Due to variations in RTTMs, caused by such as queuing delay, TCP delayed acknowledgment, or packets being routed on different paths (causing reordering), the RTT (the most current RTTM) is not used directly. To avoid rapid fluctuations, the sender maintains an estimation of the RTT, called the *Smoothed RTT* **(SRTT)** (*RFC6298*), computed using the algorithm described by *Jacobson* [1988, a21].

The smoothing works by weighting the last RTTM to $\frac{1}{8}$ on each recalculation: $SRTT = SRTT * \frac{7}{8} + RTTM * \frac{1}{8}$.

2.4.5.1 Retransmission ambiguity problem

To produce as accurate measurements of the RTT as possible, it is important to rule out ambiguous RTTMs, caused by events like packet loss or reordering.

Without TCP delayed acknowledgment, the sender can expect one ACK per packet sent. Anomalies, such as an ACK that covers two SKBs in the TCP output queue, can be an indication of packet loss. The exception would be if reordering has occured. In such circumstances, the RTTM for the lost (oldest) packet may be ignored, such that only the measurement for the last sent segment is used.

However, with TCP delayed acknowledgment enabled, both the ACK latency, and the number of ACK packets, may vary in an unpredictable manner. The data sender may deduce that TCP delayed acknowledgment is enabled on the receiver, based on fluctuations in the RTTM, but even then, it cannot know if a certain ACK was delayed or not.

In the case of a retransmitted data segment, it is difficult to know if the received ACK for that (retransmitted) segment is a reply on the original packet, that was not lost, but delayed due to TCP delayed acknowledgment, or it being routed on a different (and slower) network path (*RFC3522*). If the received ACK is in fact a reply on the original packet, and the sender uses the send-time of when the data segment was retransmitted, the calculated RTT would be smaller than the true RTT. This is what is known as the *Retransmission ambiguity problem* (*Karn* and *Partridge* [1988, a27]).

To improve the accuracy of the SRTT, Karn's algorithm suggests to ignore the RTTMs for retransmitted segments altogether in the calculation of the SRTT (*Karn* and *Partridge* [1988, a27]). For the reasons explained, *RFC6298* states that Karn's algorithm must be used when taking RTT samples.

2.4.5.2 TCP Timestamps

The TCP timestamps extension to TCP (*RFC7323*), can be used to improve the accuracy of the RTTM. By using two new optional header fields, *sender timestamp* and *echo-reply timestamp*, it is possible for the sender host to identify which sent packet resulted in a specific ACK packet. On each outgoing packet, the sender host sets the *sender timestamp* header field (`TSval`). When the receiver side replies with an ACK packet, it will set the *echo-reply timestamp* field (`TSecr`) with the value from the incomming packet's *sender timestamp* field.

RFC7323 specifies that, when using TCP delayed acknowledgment, the ACK's `TSecr` field should reflect the the oldest segment that is cumulatively ACKed. As illustrated in figure 2.5, we see that the value in the *echo-reply* field (`TSecr = 1`) of the ACK packet, reflects the timestamp (`TSval`) of segment *A*, even when segment *C* is cumulatively ACKed. This is to ensure that the sender measures the effective RTT, which includes the extra delay caused by TCP delayed acknowledgment. `TS.Recent` is a local variable on the receiver side, that always reflects the timestamp value to be used in the `TSecr` field of packets sent in return.

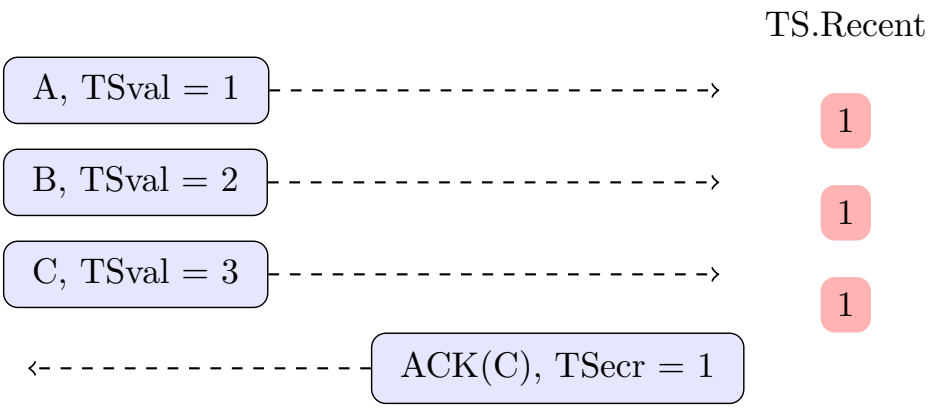


Figure 2.5: Example of TCP timestamps with TCP delayed acknowledgment (From *RFC7323*)

Based on billions of sampled connections from Google's Web servers, *Dukkipati*, *Mathis*, et al. [2011, a19] found that 12% negotiated TCP timestamps. While TCP timestamps is enabled by default in the Linux kernel, it is disabled by default in MS Windows, which explains the low amount of timestamp-enabled connections in their results.

Eifel-detection-algorithm

Instead of solving the *retransmission ambiguity problem* using Karn's algorithm, by discarding any RTTMs of retransmitted segments, TCP timestamps can be used.

The timestamp-based Eifel detection algorithm (*RFC3522*), enables a sender to unambiguously distinguish which packet an ACK is a response to. This is done by using the *sender timestamp* values of the packets (SKBs) in the TCP output queue, and the *echo-reply timestamp* of the incoming ACKs.

Figure 2.6 shows how the TCP timestamps are set when a packet is reordered. When *C* arrives before *B*, a dupACK is sent in reply. When *B* arrives, filling the hole between *A* and *C*, the receiver can cumulatively acknowledge up to *C*, causing the ACK to cover two packets. Without TCP timestamps, that would be an ambiguous situation if a retransmission for packet *B* was sent before the ACK arrived, as explained in section 2.4.5.1. However, with TCP timestamps enabled, due to the requirement that the TCP timestamps must reflect the earliest packet (with lowest sequence number) that is ACKed, the result is that the *TSecr* value of the ACK that cumulatively acknowledges *C*, reflects packet *B*'s timestamp. Therefore, the sender knows that packet *B* was not lost, but a reordering occurred,

Had *B* been lost, the ACK on *C* would be sent after the retransmission of *B* arrived. In that case, the value of the *TSecr* field would reflect the timestamp of the retransmitted *B* segment, and not the original packet.

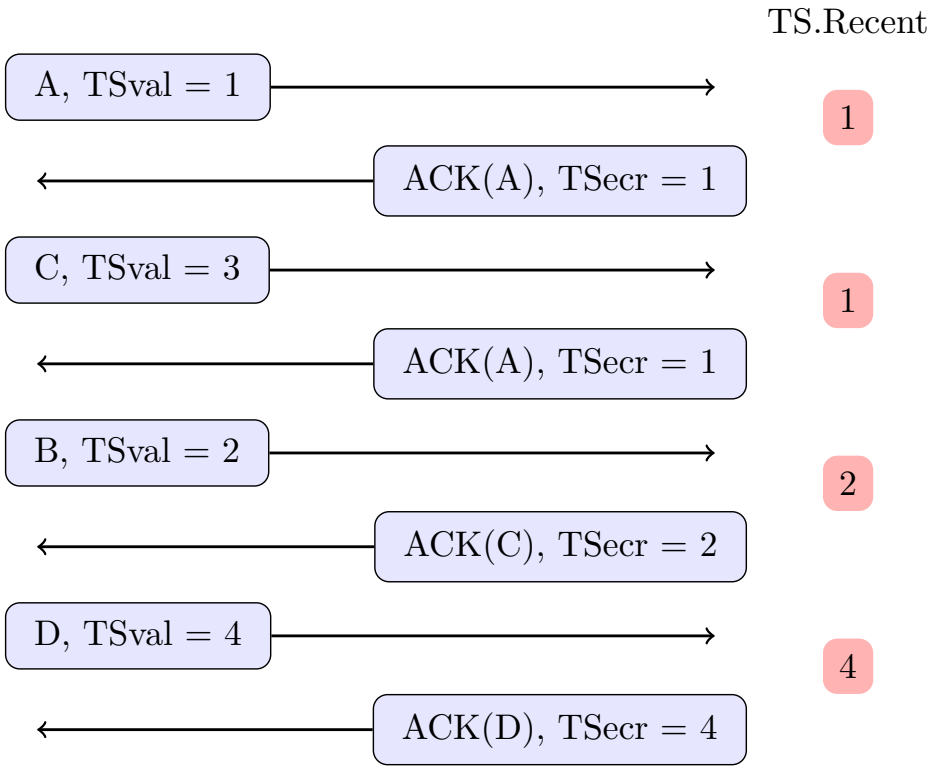


Figure 2.6: Example of TCP timestamps on packet reordering (Customization of the example from *RFC7323*)

2.4.6 Retransmission timeout

In the most basic form, TCP detects data loss by the lack of ACKs. If no ACKs arrive within a given threshold, an RTO is triggered, which indicates to TCP that the packet was lost and needs a retransmission. The greater the RTO timer value, the longer it takes for a retransmission to be initiated, and the data to arrive correctly at the receiver.

The RTO timer value is calculated so that it should only trigger when necessary. If fast recovery fails, i.e., the retransmitted data is not ACKed before the RTO is triggered, the RTO timer must be large enough to give the retransmitted packet time to be ACKed. This indicates a value greater than $RTT * 2$. The calculation must also take into account that the receiver end may have TCP delayed acknowledgment enabled which allows it to delay sending the ACK for up to 500 ms (*RFC1122*).
RFC2988 specifies the following calculation for the RTO timer:

Where G is the interrupt timer granularity in seconds, K is 4, and *Round-trip time variation* **(RTTVAR)** is the variations of the RTTMs. *RFC2988* specifies that, if the computed RTO timer is less than one second, it should be set to one second, and refer to research that suggests that a large minimum RTO timer is necessary to

$$
\begin{aligned}
SRTT &= R \\
RTTVAR &= R/2 \\
RTO &= SRTT + max(G, K * RTTVAR) \\
RTO &= MIN(1000, RTO)
\end{aligned}
\tag{2.4}
$$

avoid spurious retransmissions.

However, many TCP implementations deviate from the standard, such as the Linux kernel, where they have chosen a minimum limit of 200 ms (*Sarolahti* and *Kuznetsov* [2002, a28]), instead of the 1000 ms specified in *RFC2988*. Other OSes have the minimum RTO timer value even lower, like FreeBSD, where it is set to 30 ms (*FreeBSD 9 source code* [2014, b8]).

2.4.7 Exponential Backoff

As part of the RTO timer calculation, *RFC1122* specifies that the implementation must include exponential backoff, which means that for successive RTOs on the same segment, the RTO timer value must be exponentially increased. This makes the sender wait even longer before retransmitting packets that have not been ACKed, which is meant to have a positive effect on a congested network. However, performance analysis on the exponential backoff algorithm has shown contradictory results (*Kwak*, *Song*, and *Miller* [2005, a29]). *Mondal* and *Kuzmanovic* [2008, a30] even argue to remove the exponential backoff algorithm from TCP altogether, claiming that it can be done without causing instability issues in the Internet.

A time-dependent thin stream will contribute less to congestion than a greedy stream, due to the relatively few packets it sends. Therefore, the results of the exponential backoff can be said to cause a disproportionate decrease in the performance (increased latency) for thin streams, relative to the amount of traffic and congestion produced compared to a greedy stream.

2.5 Fairness

While the primary goal of the CC algorithms is to reduce congestion, and ultimately prevent congestion collapse, the algorithms should strive for fairness. In short, the available network resources should be fairly distributed among the users. Despite of how simple it sounds, the topic of network fairness is a complex one.

Any evaluation of network protocols, being changes to existing protocols, or proposals of new protocols, must consider the key issue of fairness. One must consider how fair it is against other streams, using the same, or similar protocols, or a mix of different protocol types.

2.5.1 Measuring fairness

A popular way of measuring fairness in networks is Jain's fairness index (equation 2.5) (*Bhatti* et al. [2008, a7]). While the index is generic and may be used with any kind of resource, a common metric is throughput. The fairness index is calculated by using the end-to-end throughput for all streams that share a path, or parts of a path, through the network. The fairness index ranges from 0 to 1, where the index is 1 when all streams are allocated an equal share.

$$Fairness = \frac{(\sum_{j=1}^{n} S_j)^2}{(n\sum_{j=1}^{n} S_j^2)} \tag{2.5}$$

The throughput-based fairness-index applies only to greedy streams of infinite length. What is not considered is when network streams have different requirements, and different lengths. If a stream is satisfied with less than what the network could maximally provide, the fairness index states that the resource allocation is unfair (*Fuchs* [2014, a18]).

If the fairness measure index can not consider the different requirements for different streams, how can it tell what is fair? It is a major issue when we consider greedy streams competing with thin streams, where widespread CC algorithms measure fairness by throughput, and primarily try to optimize the throughput for a maximum link-utilization.

2.5.2 Fairness metrics

There is no agreement in the network community on the goals for measuring fairness (*RFC5166*), understandably, as it all depends on the use case and applications. There are a multitude of fairness metrics that apply to different applications (use-cases) to a varying degree. Some relevant metrics for thin streams are:

- **Throughput** is the most common fairness metric, which can be measured on the network routers as link-utilization and as per-connection transfer times (e.g. the total time used to transfer a file). The throughput may also be distinguished from the goodput as the useful data coming through the network.

- **Delay** can be measured on the routers in the network as queuing delay, or per flow as packet delay, or per data segment which includes retransmissions. It could also be measured as the application layer latency, the time from the data was sent to the kernel on the sender, to when it enters the (user-space) application on the receiver side. While thin streams care about the per-packet latency, greedy streams only care about the per-packet latency in regards to the effect it has on the per-connection transfer time.

- **Packet loss rates** can be measured either for the network as a whole or per stream or flow. This can include congestion events such as *Explicit Congestion Notification* **(ECN)**. Greedy streams care about packet loss only

in regards to how it effects the per-connection transfer time, while time-dependent thin streams are more likely to care about every lost packet due to the increased latency for the data-segment in the lost packet.

- **Robustness to misbehaving users** considers how well the CC algorithm handles misbehaving users that try to exploit or bypass the algorithm to gain an advantage over competing streams.

2.5.3 TCP-Friendliness

TCP-friendliness is how well a network stream plays with a TCP stream. By definition, TCP is TCP friendly, but the different TCP-variants are not equally friendly, so apparently there are degrees to the friendliness.

With more and more protocols appearing in competition with TCP, *TCP-friendliness* has arisen in the network community as a goal for behavior of network streams. Considering the success of TCP on the Internet, it is understandable that the network community is reluctant to change the rules.

With the continuous development of TCP until today, where different properties and characteristics of the TCP streams changes, evaluating TCP-friendliness is not a precise science. *Floyd* and *Fall* [1999, a22] define it as: "a flow is TCP-friendly if its arrival rate does not exceed the arrival of a conformant TCP connection in the same circumstances."

The reason *most* such definitions include a mention about similar *conditions* or *environment*, is that TCP, or at least its earlier versions, are not completely fair against competing TCP streams with differing RTTs. This has been a well known problem with the CCs like TCP New Reno and those before, where *Floyd* and *Fall* [1999, a22] show, by simulation, that the the throughput of a stream using these CCs varied inversely proportional to the connections RTT. When two TCP streams with differing RTTs competes over the same bottleneck, the high-RTT stream will lose due to this.

This is what TCP BIC, that TCP Cubic builds upon, addressed. By letting the growth rate depend on the time between two consecutive congestion events, defined as when it enters fast recovery, TCP BIC's growth function is *independent* of the RTT. Therefore, the two streams in the previous example will develop approximately the same window size, even when they have different RTTs.

A TCP friendly stream must react similarly to TCP when loss occurs within a window, by reducing its CWND to at least half, and increase the CWND by a constant rate of one packet per RTT at most. This AIMD response gives a maximum throughput described by equation 2.2 which must not be exceeded.

2.5.4 Fair allocation of what among what?

An issue with the fairness maintained by the AIMD algorithm, known as flow-rate fairness, is that it treats all streams in a network as separate entities.

Consider a scenario with a network with five hosts sharing a bottleneck gateway, where each host has one TCP stream (that requests the same amount of bandwidth) where the sum of requested bandwidth for all streams surpasses the bottle-

neck rate. According to flow-rate fairness the hosts would be getting a fair share of the network resources, when over time, the rate of the streams converges against an equilibrium (*Jacobson* [1988, a21]).

However, if one of the hosts uses four TCP streams instead of one, that one host will get half of the network resources instead of $\frac{1}{5}$. Simply by using multiple connections, a host can gain an unfair advantage over the other hosts, some would say, but the flow-rate fairness is still maintained in this scenario.

This is described in-depth by *Briscoe* [2007, a31], where the entire idea of flow-rate fairness is "debunked" - flow-rate fairness does even answer the right questions, allocate the right thing or allocate between the right entities, they say.

2.6 Mechanisms for improving Latency

Historically the development on TCP's CC has focused mainly on improving throughput performance on network links that have become faster and faster by the years (*Stewart* et al. [2011, a9]).

The main reasons why thin streams are suffering when used with TCP can be attributed to a following:

- **The few PIFs** cause the thin streams to not trigger dupACKs as fast as greedy streams. This means that it takes longer for them to initiate retransmits for lost packets.
- **Due to TCP's in order guarantee**, one lost packet will cause a HOL blocking which results in severe increase in the application layer delay.
- **The exponential increase of the RTO timer** will in cases where retransmissions are lost, due to severe congestion, or simply bad luck, increase the latencies for thin streams.
- **AIMD's the harsh reaction to loss** by halving the CWND after a fast retransmit hits thin streams hard.

In recent years, some work has focused primarily on improving the latency for TCP streams. We will look closer at some of these mechanisms:

- *Linear Retransmission Timeout* **(LT)**
- *Modified fast retransmit* **(mFR)**
- *Early retransmit* **(ER)**
- *Tail loss probe* **(TLP)**
- *RTO Restart* **(RTOR)**
- *Redundant Data Bundling* **(RDB)**

Petlund [2009, a4] investigate how to improve the latency for time-dependent thin streams using TCP. They implement three modifications to the Linux kernel, that in various ways attempt to improve the latency; LT, mFR, and RDB.

To limit how aggressive the mechanisms are, and ensure they only affect thin streams, LT and mFR are only used if the stream is considered *thin*, as defined by the function `tcp_stream_is_thin` (see code listing 2.1). This function verifies that the connection has less than four PIFs, and is not in the initial slow-start phase.

Linux kernel source

```
1 /* Determines whether this is a thin stream (which may suffer from
2  * increased latency). Used to trigger latency-reducing mechanisms.
3  */
4 static inline bool tcp_stream_is_thin(struct tcp_sock *tp) {
5     return tp->packets_out < 4 && !tcp_in_initial_slowstart(tp);
6 }
```

The function in the Linux kernel that determines if a stream is thin

Code Listing 2.1: `tcp_stream_is_thin` in net/tcp.h

2.6.1 Linear Retransmission Timeouts

The exponential backoff mechanism will double the RTO timer each time an RTO is triggered. First when an ACK is returned for a segment that was not retransmitted, will the RTO timer be recalculated. This is because it can be unclear which data packet was ACKed - the initial data packet or one of the retransmissions. As the RTO timer calculation depends on the RTT, the calculation is delayed until the reception of an ACK which gives a good RTT estimation.

To improve the situation for thin streams, the LT option disables exponential backoff for up to 6 timeouts (RTOs) as long as the stream is thin. On the seventh RTO exponential backoff is resumed. When LT is triggered, instead of being doubled, the RTO timer is set to the SRTT + `rttvar`, which is the smoothed maximum value of the medium deviation (of the RTT) within the last RTT period.

Figure 2.7 illustrates the difference between LT and exponential backoff. As this mechanism will have effect only when retransmissions are lost, it will have the most effect in heavily congested scenarios where subsequent RTOs occur. However, it will also have effect when subsequent RTOs are lost due to bad luck.

The LT modification was included in Linux kernel version 2.6.34 (*Petlund* [2010a, b1]).

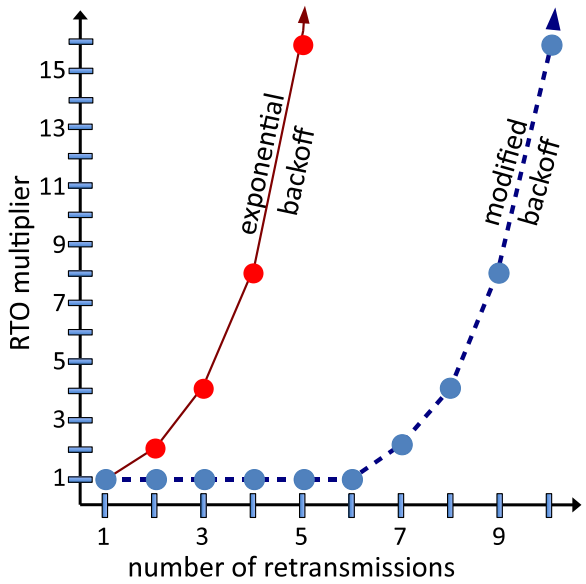


Figure 2.7: Illustration of the Linear Retransmission Timeouts mechanism. (*Petlund* [2012, a32])

2.6.2 Modified fast retransmit (mFR)

By default fast recovery is triggered when three dupACKs have been received. In the case of thin streams, the high *inter-arrival time* **(IAT)** makes it more likely that an out-of-order packet is caused by packet loss. With three dupACKs required to trigger fast recovery, it requires the sender to send three packets after a packet is lost to generate the three dupACKs.

Subfigure 2.8.(a) illustrates how the loss of one packet affects a stream when fast retransmit is triggered after three dupACKs. When packet P1 is lost, the receiver will retransmit the packet when receiving the third dupACK triggered when P4 is received. The consequence is that packets P2-P6 must be delayed on the receiver side before being delivered to the application layer.

With thin streams, that often have very few PIFs, the fast retransmit may never be triggered. This is also an issue for short-flows, such as HTTP transactions, that only send few packets in total. Tests on Google's Web servers showed that fast recovery accounted for 25% of the retransmission on short-flows, and 50% on bulk video traffic (*Dukkipati*, *Mathis*, et al. [2011, a19]), which means that a surprisingly large amount of the retransmissions are triggered by RTOs. *Balakrishnan* et al. [1998, a33] also find similar results for HTTP traffic, where more than 50% of the retransmissions by a busy web server are triggered by RTOs.

The mFR mechanism was developed to improve the latency for thin streams in such cases. With the `tcp_thin_dupack` option enabled, the required number of dupACK to trigger a fast retransmit, is reduced from three to one for thin streams.

Subfigure 2.8.(b) illustrates what can happen when a stream sends only one packet per RTT. It does not receive the third dupACK before the RTO timer is

triggered, causing grave delays. In subfigure 2.8.(c) we see how this situation is improved using the mFR option. By triggering fast retransmit after the first dupACK fewer packets are delayed due to HOL blocking.

However, in addition to the PIF limit and the test for initial slow-start phase, there are other criteria that restricts when mFR may trigger a fast retransmit. In code listing 2.2 we see the code that tests if mFR may trigger a fast retransmit. In addition to calling `tcp_stream_is_thin`, and testing for SACK support, it also requires no unsent data being available in the TCP output queue, as well as the return value of `tcp_dupack_heuristics` being more than 1. With FACK enabled by default in the kernel, this return value is `tp->fackets_out`, the number of outstanding segments according to the FACK accounting.

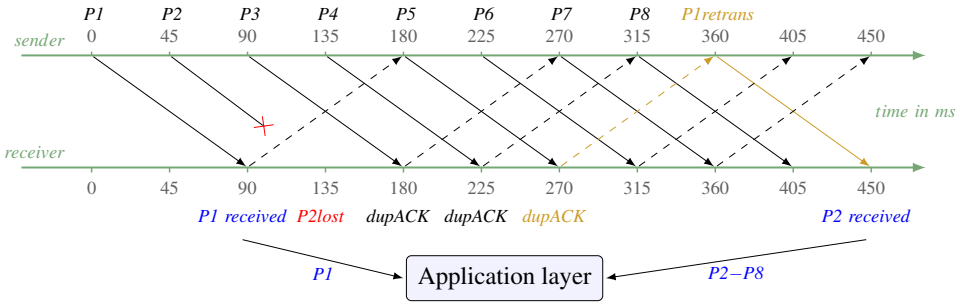
Linux kernel source

```
1 if ((tp->thin_dupack || sysctl_tcp_thin_dupack) &&
2      tcp_stream_is_thin(tp) && tcp_dupack_heuristics(tp) > 1 &&
3      tcp_is_sack(tp) && !tcp_send_head(sk)) {
4      return true;
5 }
```

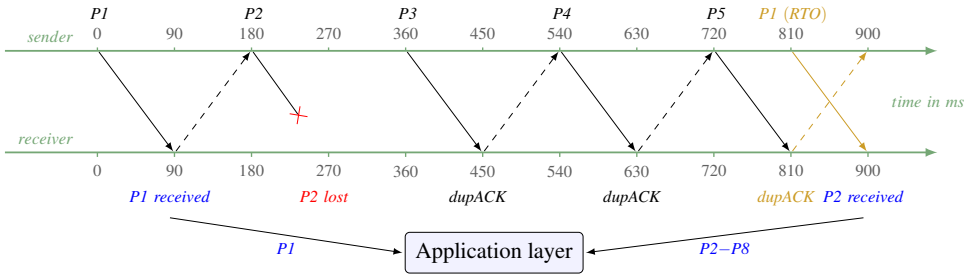
The code in the Linux kernel that tests if mFR can be used for a stream.

Code Listing 2.2: The code that tests if mFR should be used.

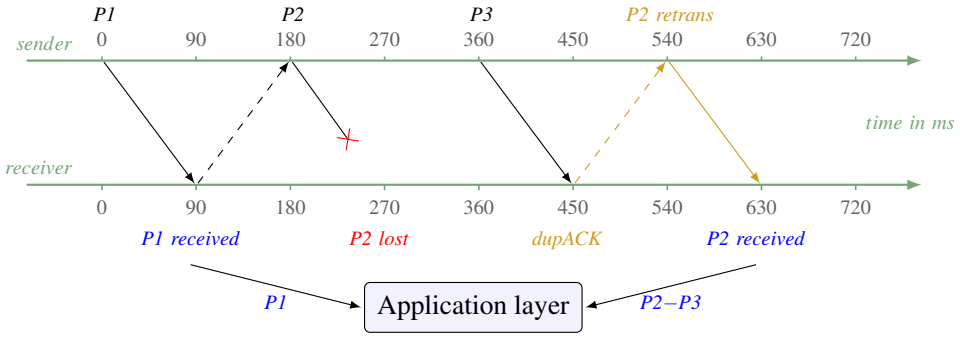
The mFR mechanism was included in Linux kernel version 2.6.34 (*Petlund* [2010b, b2]).



(a) Fast retransmit triggered after three dupACKs



(b) Fast retransmit is first triggered after an RTO is triggered



(c) Fast retransmit triggered after one dupACK

Figure 2.8: Timelines showing when fast retransmit is triggered

2.6.3 Early retransmit

ER is a mechanism for TCP and SCTP (*RFC5827*), that is similar to mFR in that it tries to improve the latency for streams that do not receive enough dupACKs to trigger a fast retransmit. Like mFR, it allows the sender to lower the threshold for the number of required dupACKs to trigger a fast retransmit. Where mFR allows a fast retransmit on the first dupACK as long as there are less than four PIFs, the segment based ER implemented in the Linux kernel allows a fast retransmit only when *a)* the flight size is less than four, and *b)* there is either no unsent data in the TCP output queue, or the advertised RWND does not allow new segments.

ER was included in the Linux kernel version 3.5 (*Cheng* [2012, b9]) and is enabled by default (*Linux kernel IPv4 variables* [2014, b10]).

If the mFR mechanism is enabled, ER is automatically disabled by the Linux kernel.

2.6.4 Tail Loss Probe

TLP is designed to help in cases where losses occur at the end of a packet burst (*Dukkipati*, *Cardwell*, et al. [2012, c23]). If the last packet or packets are lost, the sender will never receive a dupACK, and is forced to wait for an RTO before retransmitting. TLP will send probe segments to get extra ACKs from the receiver, and in the case of loss, dupACKs. A probe segment is sent when the probe timeout (PTO), which is always set to be lower or equal to the RTO timer, signals that an ACK is overdue, i.e., should have been received, and if the connection satisfies the criteria: *a)* there is outstanding unacknowledged data, and *b)* it is otherwise idle, i.e., not receiving any ACKs or is limited by CWND/RWND or is application limited.

It is also a requirement that the connection has negotiated SACK to use TLP. If there there is any unsent data available, and the RWND allows it, the probe segment will send new data. Otherwise, the probe segment will retransmit the most recently sent segment.

TLP was included in the Linux kernel version 3.10 (*Dukkipati* [2013, b3]) and is enabled by default (*Linux kernel IPv4 variables* [2014, b10]).

2.6.5 RTO Restart

RTOR is a mechanism for TCP and SCTP that provides faster loss recovery for connections with small amounts of outstanding data, such as short-lived or application limited connections *Hurtig* et al. [2014, c24]. It specifically aims to improve the latencies in situations where fast retransmit cannot be used.

The standard algorithm for managing the RTO timer specifies that it should be reset on each incoming ACK (*RFC6298*), which in the worst case scenario can give a loss recovery time of $RTO + RTT + delACK$, where $delACK$ is additional delay on the ACK added by TCP delayed acknowledgment.

RTOR suggests to lower the RTO timer by setting it to $RTO - T_{earliest}$, where $T_{earliest}$ is the time elapsed since the earliest of the currently outstanding segments was sent.

This alternative RTO timer calculation should only be used when the number of outstanding segments is less than four.

2.6.6 Redundant Data Bundling

RDB is a technique which aims to reduce the latency by resending data even before a packet loss is detected. RDB utilizes the free space in packets that have less payload than the MSS, by bundling older un-ACKed data onto packets with new.

Figure 2.9 shows an example of an Ethernet frame containing a TCP segment with less data than allowed in a full MSS. By using the "free" space in the ethernet frames, one can argue that RDB does not generate extra workload for the nodes in the network. The idea is that when a packet is lost, the data is bundled with the next packet that may already be on its way. This way of bundling data can be regarded as a retransmission mechanism that retransmits data prior to detecting packet loss, unlike regular retransmission mechanisms, that retransmit based on indications that loss has occurred.

While TCP can handle packet reordering, and, with the help of sequence numbers, is able to order the data properly before passing it to the application layer, the data in each TCP packet must be sequential. This constraints the RDB mechanism to only bundle the data of adjacent SKBs that are earlier in the TCP output queue.

Figure 2.10 illustrates how the data is bundled in the TCP segments.

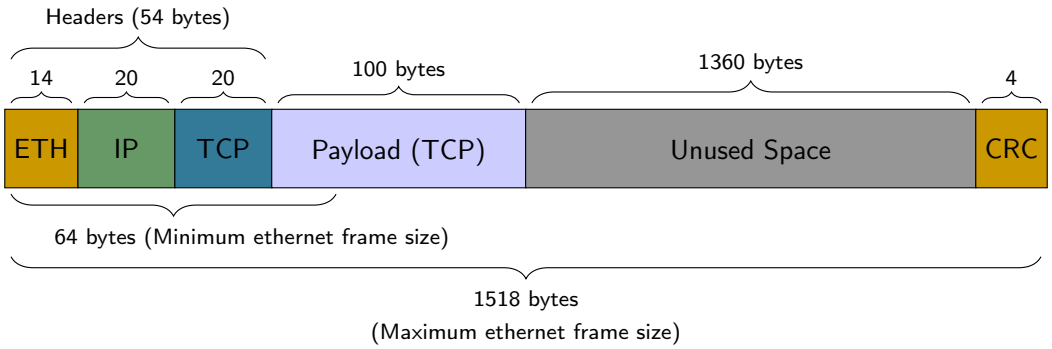
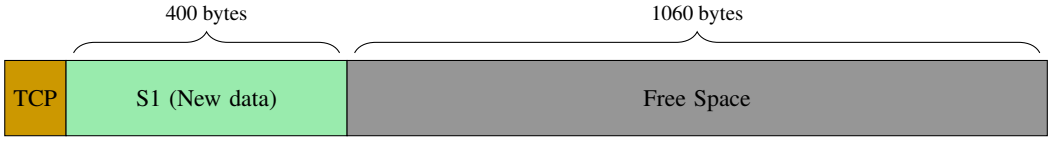


Figure 2.9: Example of an Ethernet frame for a TCP packet with 100 bytes payload. The headers depicted contain only the necessary information with only the mandatory TCP options.

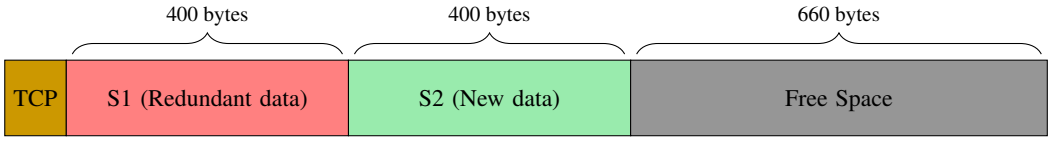
In figure 2.11 we see a short packet sequence timeline, where a thin stream using RDB, avoids retransmissions due to redundant data being bundled with the segments containing new (unsent) data.

Sender side modification

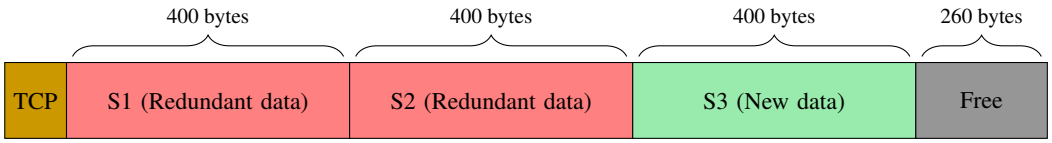
What makes RDB somewhat special, is that it changes significantly how the data is transferred by TCP, while still being a sender-side modification only. No changes are necessary on receiver hosts to be able to handle the RDB packets, which makes it easy to deploy and use in today's networks.



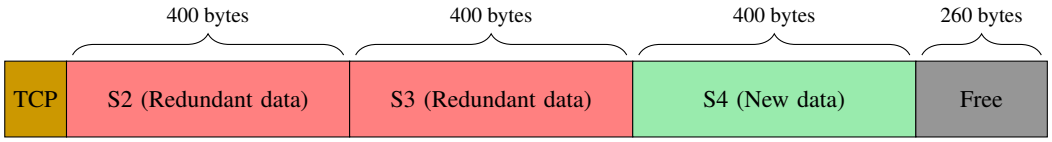
(a) TCP segment containing 400 bytes of new data



(b) TCP segment containing 1 segment of redundant data (S1) and 1 segment of new data (S2)



(c) TCP segment containing 2 segments of redundant data (S1, S2) and 1 segment of new data (S3)



(d) TCP segment containing 2 segments of redundant data (S2, S3) and 1 segment of new data (S4)

Figure 2.10: Examples showing how RDB bundles the data of previously sent packets onto packets with new data.

In an MMORGP scenario, where multiple clients communicate with servers hosted by the game company, it will be beneficial even if only the game server uses RDB. The traffic sent from the server will benefit from RDB, while the unmodified TCP traffic from the clients will not.

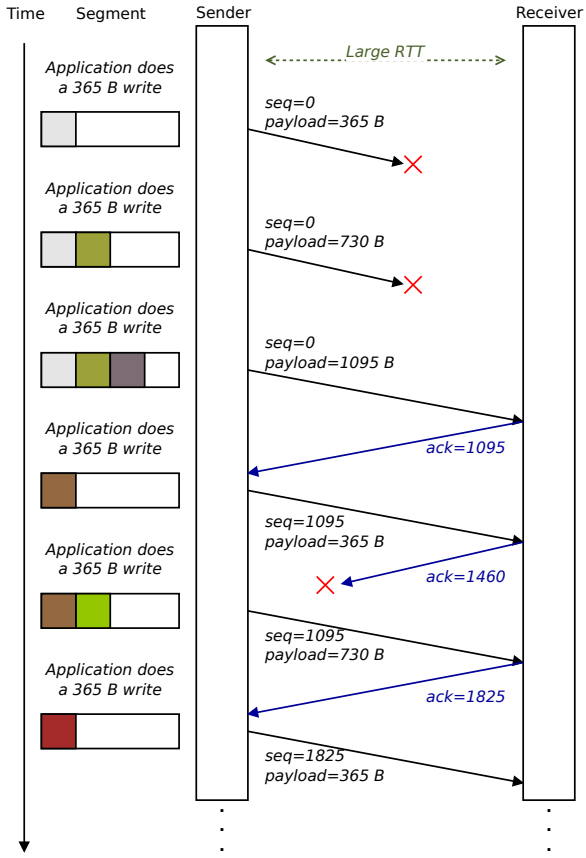


Figure 2.11: Packet timeline of a TCP stream with RDB (*Markussen* [2014, a34])

2.7 RDB prototype v1

RDB started out as an idea in *Søgård Paaby* [2006, a35], to extend the `retrans_collapse` function in the Linux kernel. The function `tcp_retrans_try_collapse` in `tcp_output.c` is used to collapse (merge) SKBs in the TCP output queue before retransmition. It will try to merge as many SKBs as possible. In the Linux kernel version 2.6.15, which they were studying, the function `tcp_retrans_try_collapse` was only called once to merge two adjacent SKBs, so the idea was to modify this to do this multiple times, as the current Linux kernel code does.

Building upon this idea, *Evensen* [2008, a5] implemented a more aggressive bundling mechanism that also bundles old data when transmitting packets with new data. These modifications, which we will refer to as the RDBv1, modifies the buffers in the TCP output queue by prefixing the payload of previous buffers to each new buffer. RDBv1 was implemented in the Linux kernel version 2.6.22.1 and later ported to version 2.6.23.8 (*Evensen* [2008, a5]).

The RDBv1 patch, which is included in code listing C.1, was sent to the Linux net-dev mailing lists in 2009 for feedback. No more work was done on

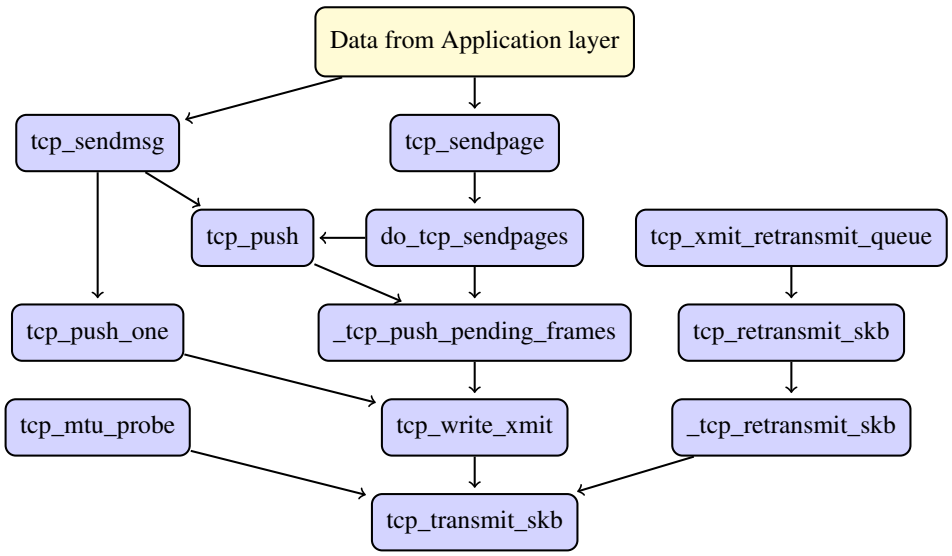


Figure 2.12: Call graph of parts of the TCP engine in the Linux kernel

the RDBv1 patch after that. The implementation details will be breifly described in section 2.7.2.

2.7.1 TCP-engine in Linux

The part of the Linux kernel that implements TCP is called the TCP engine. The main parts of the TCP engine code in the Linux kernel is implemented in the following files:

- **`tcp.c`**
  Contains initialization code for a TCP connection and the entry points for data comming from user space.

- **`tcp_input.c`**
  Handles incomming packets and processes the events that are triggered by ACKs.

- **`tcp_output.c`**
  Processes outgoing packets contained in the TCP output queue and sends the SKBs to the IP-layer for transmission.

- **`tcp_cong.c`**
  Contains the TCP New Reno CC code as well as the base code for the Linux CC framework.

- **`tcp_<CC>.c`**
  Contains alternative CC implementations, (where *CC* is replaced with) such as TCP Cubic, `vegas`, `highspeed`, `westwood` and `veno`.

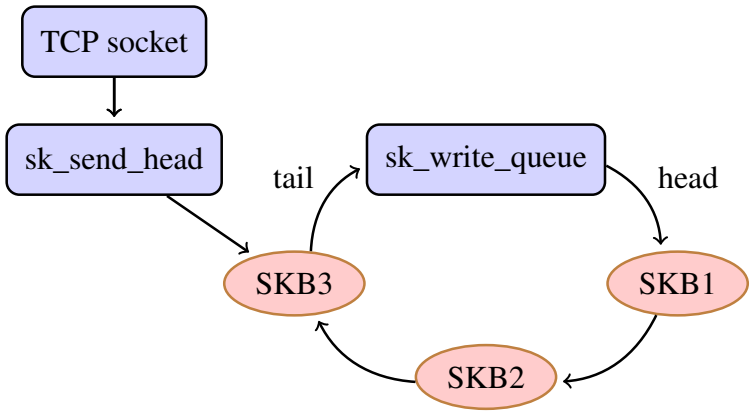


Figure 2.13: The TCP output queue

Figure 2.12 illustrates the parts of the TCP engine implemented in `tcp.c`, where data enters from user space, and `tcp_output.c`, where it is sent to the IP layer.

SKBs and the TCP output engine

The TCP output engine stores the outgoing data for each socket in a linked list called the TCP output queue as shown in figure 2.13. The data to be transferred is stored in SKBs which is a `struct` containing next and previous pointers as well as the necessary information needed to build the network packets.

The TCP output queue has two functions: *a)* to handle the send rate mismatch between the application layer and TCP, and *b)* to store the sent packets, in case of a needed retransmission, until they are acknowlegded.

The elements are ordered by the sequence numbers where `sk_write_queue` points to the first list element that contains the oldest un-ACKed data, whereas `sk_send_head` points to the data that has not yet been sent. If they point to the same buffer, no un-ACKed data is present in the queue. If only `sk_send_head` is `NULL`, all the data in the queue has been sent and is waiting to be acknowledged.

When ACKs are received, the function `tcp_clean_rtx_queue` is called to traverse the TCP output engine to remove SKBs if possible.

2.7.2 RDB prototype 1 (RDBv1)

As depicted in figure 2.14, data from the application layer enters the TCP engine in `tcp_sendmsg`. `tcp_sendmsg` is modified to call the function `tcp_trans_merge_prev` after the new data has been inserted into the TCP output queue. `tcp_trans_merge_prev` does the work of modifying the new SKBs by placing older data into the beginning of the data area. For the incomming packets, `tcp_clean_rtx_queue` is modified to clean up after the changes made to the TCP output queue, as shown in figure 2.15.

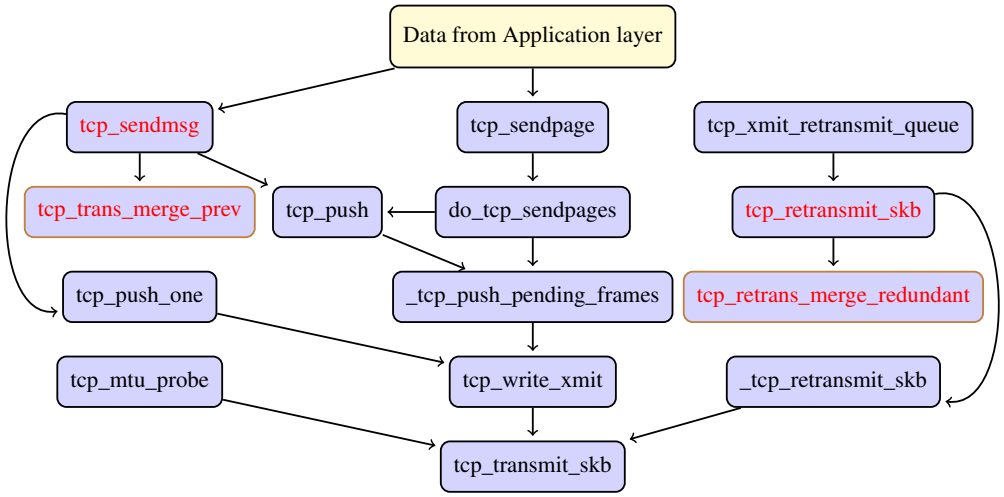


Figure 2.14: The call sequence for outgoing data in the RDBv1 prototype implementation. Nodes with red text are functions that were modified. Nodes with brown border are new functions.

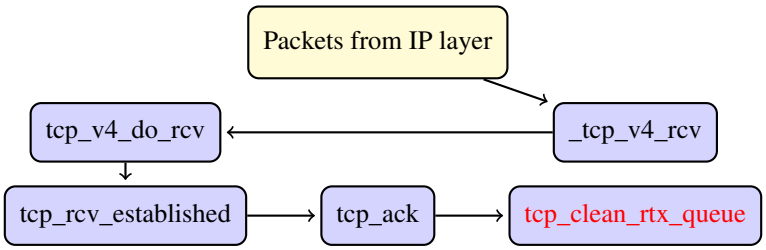


Figure 2.15: The call sequence for incomming packets in the RDBv1 prototype implementation. The node marked with red text is modified.

A prerequisite for the RDB technique, is that the receiver end will check the incomming packets by using the end sequence number, and not the start sequence number. If only the start sequence number was to be compared with the expected incomming sequence number, it would classify RDB packets as retransmitted packets and ignore them.

The function in the Linux kernel's TCP engine that handles incomming data packets, `tcp_stream_is_thin`, is shown in code listing 2.3.

In code listing 2.3 line 10, we see that it first checks if the starting sequence number is in sequence by testing if it equals `rcv_nxt`. If that is not the case it will check if the incomming SKB contains new data, by testing if `end_seq` is greater than the `rcv_nxt` (line 10). This ensures that the parts of the data that is new in an RDB packet is still processed even the when start sequence number is not the expected sequence number for new data.

Linux kernel source

```
1  static void tcp_data_queue(struct sock *sk, struct sk_buff *skb)
2  {
3      if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
4          // This is an in sequence segment
5          ....
6  queue_and_out:
7          ....
8          return
9      }
10     if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
11         /* A retransmit, 2nd most common case. Force an immediate
               ack. */
12         // Does not contain any new data
13         ....
14 out_of_window:
15         tcp_enter_quickack_mode(sk);
16         inet_csk_schedule_ack(sk);
17         return
18     }
19     /* Out of window. F.e. zero window probe. */
20     if (!before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt +
           tcp_receive_window(tp)))
21         goto out_of_window:
22     tcp_enter_quickack_mode(sk);
23
24     if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
25         /* Partial packet, seq < rcv_next < end_seq */
26         // This is where RDB packets containing both new and old
               data are handled
27         ....
28         tcp_dsack_set(sk, TCP_SKB_CB(skb)->seq, tp->rcv_nxt);
29         ....
30         if (!tcp_receive_window(tp))
31             goto out_of_window;
32         goto queue_and_out;
33     }
34 }
```

The initial tests in `tcp_data_queue` on the incoming data.
Four punctuations indicate code lines that were removed. Comments added for clarity are marked in this color

Code Listing 2.3: Excerpt from the function `tcp_data_queue` in `tcp_input.c`

2.7.3 Issues and critique of RDB

Concerns have been raised as to the aggressiveness of RDB, and how it could potentially affect other competing network streams. The RDBv1 has a sysctl option (`tcp_rdb_max_bundle_bytes`) that limits the number of bytes to bundle for each packet, but it does not use the `tcp_stream_is_thin` employed in mFR and LT to bundle only when a stream has less than 4 PIFs.

Concerns were also raised about how the SKBs in TCP output queue are modifed by adding old data to new SKBs. These complex changes to the SKBs, which are performed when new data enters the kernel, as well as before retransmitions, may lead to issues like silent data corruption, and would require an audit for every

place where changes to the data are made to be sure the data is always correct (see email D.1).

2.8 Summary

In this chapter we have introduced thin streams and interactive applications that produce network streams with thin-stream characteristics. We have looked at key properties of TCP, as well as different types of CCs, and how they relate to measuring fairness. Further, we briefly discussed some TCP mechanisms implemented in the Linux kernel aimed at improving the latency for thin streams, and finally introduced the RDB mechanism and described the RDBv1 implementation.

In light of the concerns raised about the RDB mechanism's aggressiveness from section 2.7.3, we will explore how to improve the situation for thin streams further in chapter 3.

Chapter 3

Improving the latency for thin streams

In this chapter we look at the how thin streams perform on the current thin-stream mechanisms in the Linux kernel. We confirm the performance issues of thin streams with a set of preliminary experiments, before we go into detail about RDB, and identify the key issues with the RDBv1 implementation that we want to solve.

After reviewing the state of the RDBv1 we decided on a few goals on what we wanted to improve:

- **TCP Fairness**
  Look at how to make RDB as fair as possible towards competing traffic while still achieving better latency.
- **Protection against abuse**
  Look at how to avoid any abuse of RDB to gain an advantage over other streams.
- **Code quality and design**
  Look at how the code can be better integrated with the Linux kernel, by separating as much of the code as possible out of the base code of the TCP engine.

3.1 Experiments with mFR, LT and ER+TLP

We have performed a set of preliminary experiments with the current mechanisms in the Linux kernel aimed at reducing latency for thin streams. We have used two sender hosts, one sending thin streams, and one sending greedy traffic, with the test properties from table 3.1. The host sending thin streams is set up to run the combinations of the mechanisms LT, mFR, and ER + TLP. The testbed setup used for the experiment is explained in detail in section 5.2.

Table 3.1

Test	Figure	Name	Streams	Cong	RTT	ITT	Payload	Mechanisms
1	Figure 3.2.(a)	Thin TCP	21	Cubic	150	100:15	120	
1	Figure 3.2.(a)	Greedy	10	Cubic	150	NA	NA	NA
2	Figure 3.2.(b)	Thin TCP	21	Cubic	150	100:15	120	LT
2	Figure 3.2.(b)	Greedy	10	Cubic	150	NA	NA	NA
3	Figure 3.2.(c)	Thin TCP	21	Cubic	150	100:15	120	DA
3	Figure 3.2.(c)	Greedy	10	Cubic	150	NA	NA	NA
4	Figure 3.2.(d)	Thin TCP	21	Cubic	150	100:15	120	LT/DA
4	Figure 3.2.(d)	Greedy	10	Cubic	150	NA	NA	NA
5	Figure 3.2.(e)	Thin TCP	21	Cubic	150	100:15	120	ER+TLP
5	Figure 3.2.(e)	Greedy	10	Cubic	150	NA	NA	NA
6	Figure 3.2.(f)	Thin TCP	21	Cubic	150	100:15	120	LT/ER+TLP
6	Figure 3.2.(f)	Greedy	10	Cubic	150	NA	NA	NA
7	Figure 3.2.(g)	Thin TCP	21	Cubic	150	100:15	120	DA/ER+TLP
7	Figure 3.2.(g)	Greedy	10	Cubic	150	NA	NA	NA
8	Figure 3.2.(h)	Thin TCP	21	Cubic	150	100:15	120	LT/DA/ER+TLP
8	Figure 3.2.(h)	Greedy	10	Cubic	150	NA	NA	NA

The ITT of 100:15 means that the ITTs are generated by a function that produces a uniform distribution with an average of 100 and a standard deviation of 15. The greedy streams are denoted with an ITT and payload of "NA" i.e., not applicable.

Table 3.1: Thin stream modification test setup

The plots show the aggregated ACK latency [1] for thin streams versus greedy streams (See section 5.1.1.1 for details about why and how the ACK latency is measured).

Figure 3.1 shows the first test where all mechanisms are disabled. The orange line indicates the range of extra delay for the thin-stream packets caused by queuing delay in the bottleneck. With an RTT of 150 ms added by `netem`, the minimum ACK latency is 150 ms which is why the x-axis starts at 150.

Thin stream packets with ACK latencies that fall after the sharp curve (around the 250 ms mark), have been lost. This is also indicated by the blue line being almost horizontal between 250 and 350 on the x-axis, meaning no ACK latency values in that range.

The vertical dotted lines are placed on every ITT interval (100 ms). Looking closely at the intersection between the ITT-lines and the blue line, we can see small

[1] ACK latency is the time between a data segment is first sent onto the network, until an ACK for the data segment is received.

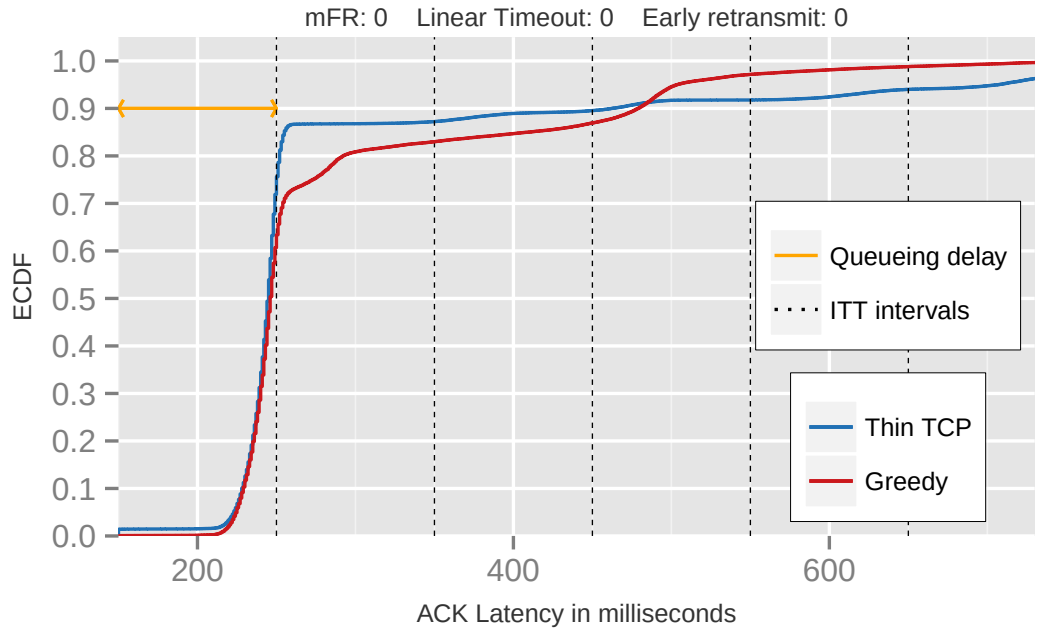


Figure 3.1: Plot of ACK-latencies showing the effect of queuing delay

curves. To get a better look at this pattern we have plotted the rest for the results (figure 3.2) using only the upper parts of the y-axis ($0.65 - 1$).

In figure 3.2 we can see how much impact packet loss can have on time-dependent thin streams. The distinct *stair pattern* in the plots illustrates how the latency of the packets are dependent on the data in earlier packets due to HOL blocking. With a loss rate of round 2.5% (see table 3.2) we see that around 10-13% of the packets are affected by increased latency due to packet loss.

The stair pattern pattern corresponds well with the vertical dotted lines that show the ITT intervals. The number of steps in the stair pattern reflects how many packets are affected by a loss, which is dependent on the relation between the RTT and ITT. In this case, where the RTT is 150 ms, and the ITT is 100 ms, we see a clear trend where up to 6 packets are affected. We go into further in details on this effect in section 5.4.2.3.

Table 3.2

Test	Name	Data packets	Retrans	Payload B	Bytes Loss %	DupACKs
1	Thin TCP	61876	1813	7.8 M	2.86	6208
1	Greedy	118361	3191	171.2 M	2.69	20671
2	Thin TCP	61881	1717	7.8 M	2.75	6010
2	Greedy	118198	3032	170.9 M	2.56	20872
3	Thin TCP	61843	1692	7.8 M	2.71	5902
3	Greedy	118513	2925	171.4 M	2.47	20416
4	Thin TCP	62048	1633	7.8 M	2.54	5741
4	Greedy	118187	2999	171.0 M	2.53	20579
5	Thin TCP	63153	1645	7.8 M	2.56	5104
5	Greedy	117997	2845	170.6 M	2.41	20094
6	Thin TCP	62232	1708	7.8 M	2.70	5115
6	Greedy	118546	2940	171.5 M	2.48	20896
7	Thin TCP	63072	1568	7.8 M	2.44	4823
7	Greedy	117906	2810	170.6 M	2.38	20242
8	Thin TCP	63114	1662	7.8 M	2.59	5066
8	Greedy	117952	2898	170.6 M	2.45	20475

Table 3.2: Thin stream modifications test results

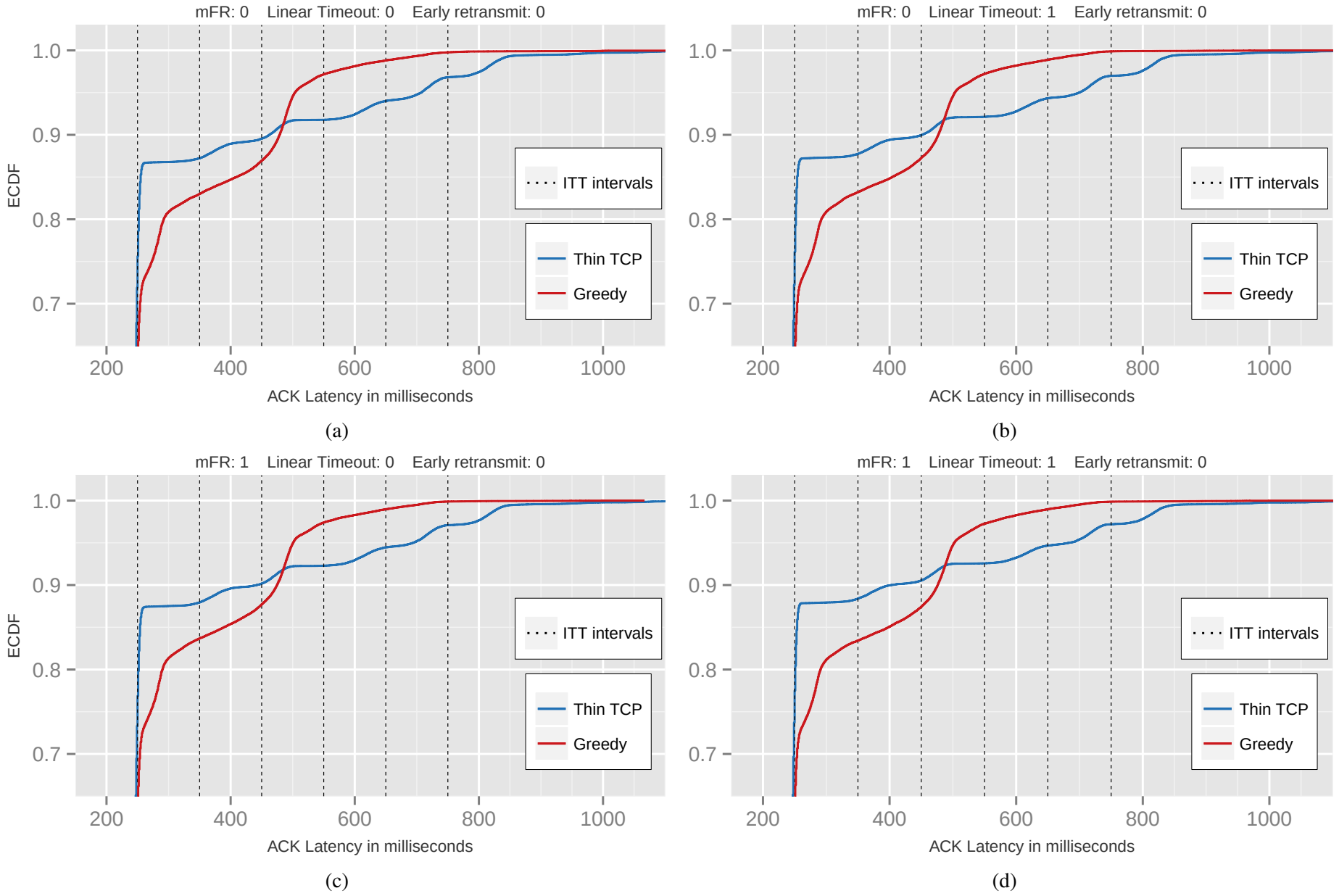
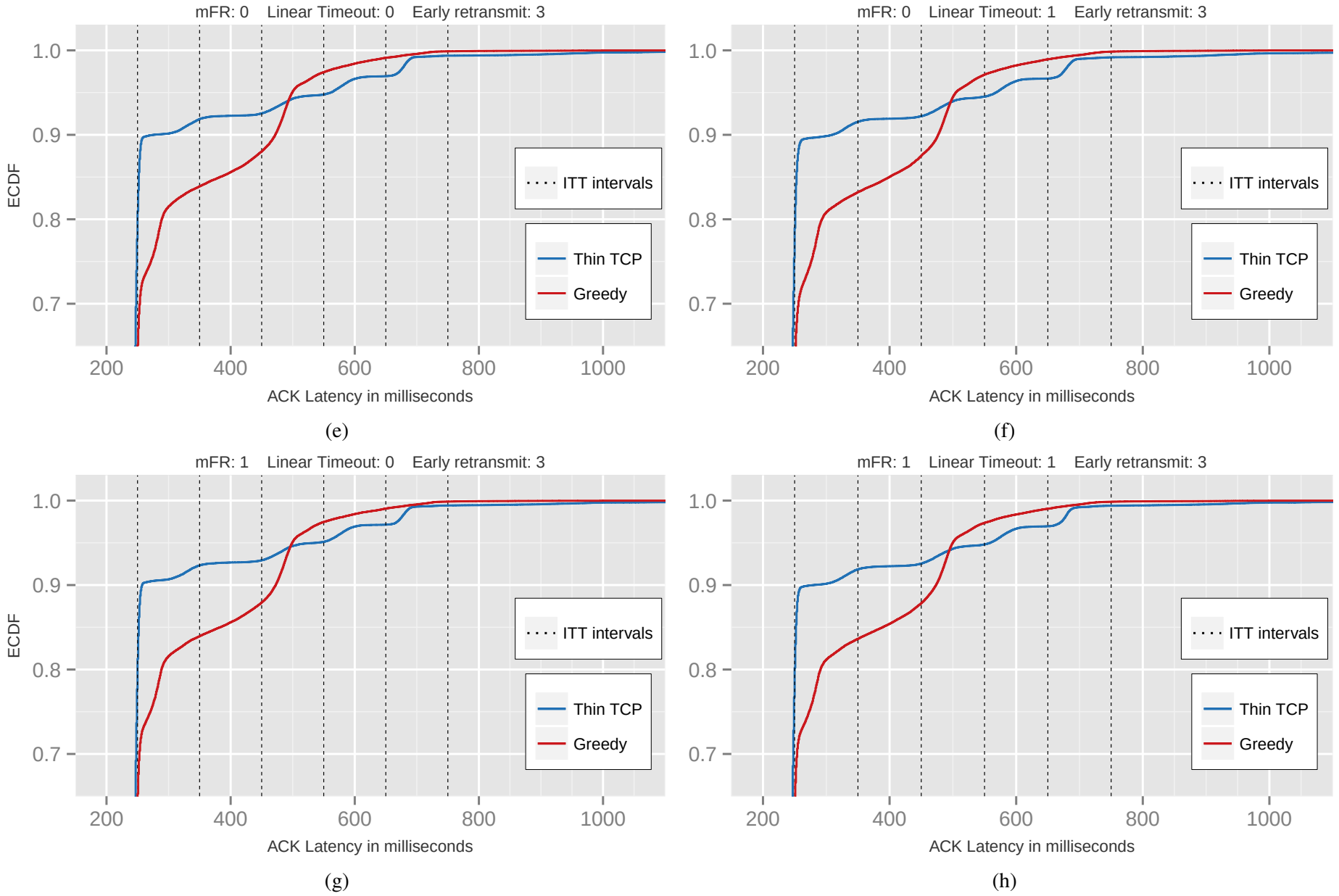


Figure 3.2: ACK latencies for thin vs greedy streams using the mFR, LT and ER mechanisms



(e) (f)

(g) (h)

Figure 3.2

From the plots with ER and TLP enabled we see that the thin-stream latencies are somewhat better, where the 90^{th} percentile is consistently at 250 ms, whereas the plots without ER and TLP have the $87-88^{th}$ percentile at 250 ms.

For LT to have an affect, it requires loss of retransmitted packets. Losing retransmissions will naturally have a very negative effect on the latency, but is also a less likely event than losing only the first transmission. Therefore, we can not expect to see considerable effect from LT in these conditions.

The goal of this experiment is not to perform an in-depth analysis on the effects of the thin-stream mechanisms, and how they compare to each other. We show these results to illustrate the problem that none of the current mechanisms solve, namely the HOL blocking effect caused by retransmission latency.

3.1.1 Head-of-line blocking

Due to the in-order guarantee of TCP, any data segments arriving after a lost packet are held back on the receiver, also known as HOL blocking. To uphold the ordering guarantee, the data that is held may not be sent to the application layer until the lost data segment has been retransmitted.

The *stair pattern* in the plots illustrates how a packet loss not only affects the latency of the lost packet, but due to HOL blocking, a range of packets coming after. These grave delays, can be alleviated by increasing the effectiveness of the traditional re-transmission mechanisms, such as mFR. However, the extra delay of minimum one RTT (for the dupACK to return and the retransmission to arrive) is still very high, which leaves great potential for improvements.

3.2 TCP Fairness and RDB

With no mechanisms to control how much and how often data is bundled, RDBv1 was deemed too aggressive, and rightfully so. We therefore started out with the task of finding the sweet spot for bundling data so that we get a best possible latency improvement without affecting competing streams negatively.

It is of vital importance that RDB can not be misused to get an advantage over other streams by ignoring any signs of congestion. At a minimum, RDB should meet the criteria that an RDB stream can not get an advantage throughput-wise over a competing greedy TCP stream.

3.2.1 RDB hiding loss events

At interesting effect with RDB is that it hides the loss of a packet, if the next packet containing the lost data is transferred successfully. The amount of loss required for TCP to detect a loss depends on the number of previous packets that are bundled. If each new data segment from the application is 700 bytes, the data of only one packet can be bundled with each sent packet. If only one packet is lost, the next packet will hide the loss, but if two packets in a row are lost, the data in the next packets will not be in sequence and hence trigger a dupACKs. If the data segments from the application are 400 bytes, as depicted in figure 2.10, there

is room to bundle the data of two earlier packets in the TCP output queue. If the frames in subfigure 2.10.(a) and subfigure 2.10.(b) are lost, S1 is still transmitted in subfigure 2.10.(c), but if that frame is lost as well, subfigure 2.10.(d) will cause a gap in the sequence numbers.

If an application sends segments of 100 bytes to the kernel, and the MSS is 1460 bytes, there will be room to bundle the data of 13 previously sent packets. This will require 14 lost packets in a row, for a gap in the sequence space to occur on the receiver side, resulting in a dupACK.

3.2.1.1 Active queue management

When the incoming buffer queue of a router in a network is constantly full the network is congested (at least that path in the network). When the router has no more room it simply drops any incoming packets, which is known as the tail drop algorithm, the traditional solution for the most basic routers.

Active queue management **(AQM)** is the smarter alternative to tail drop, where the queuing discipline used by the router drops packet according to some algorithm.

Explicit Congestion Notification

Some routers with AQMs may also support ECN, an extension to the TCP and IP protocols. In an ECN scenario, a router that is not yet congested, but finds the traffic flow becoming too heavy, may set the ECN flag on packets before forwarding them on the correct path. This will work with regular TCP as well as with an RDB stream, as the ECN mark will be detected by the receiving end host which echoes back the ECN mark on the ACK-reply. On receiving the ACK, the sending host will be able to do adjustments to the send rate accordingly.

Loss based AQMs

Queuing disciplines like *Random early detection* **(RED)**, works by dropping incoming packets before the router is too congested. This induced loss is a congestion indication signal to the sender node that there is congestion, so that it can make adjustments to prevent overflowing the network.

For RDB streams we encounter a problem with these types of AQMs. A TCP stream will react to the loss of a single packet, but with an RDB stream, the loss would be hidden by the redundant data in the next packet resulting in the congestion signal from the AQM being completely ignored. In this case RDB would get an advantage over other streams.

3.2.2 Abusing and misusing RDB

The potential of abusing RDB to gain an advantage over other network streams, or to sabotage for other streams on the network, is significant. As RDB hides some of the loss from the TCP engine, the CWND could grow faster than a regular TCP stream and not back off as fast as other streams would when they detect loss.

RDB is meant to help thin streams, but when used on a greedy stream where the application adapts the send calls to the kernel to improve the potential of bundling redundant data, it could get an advantage throughput-wise. Therefore, care has to be taken to prevent any abuse of RDB by applications that do many send calls with less than half an MSS per call.

The kernel receives the data to be transferred through the system call send. Normally the kernel will delay the sending of the data to avoid smaller packets being transmitted. This is because of the Nagle's Algorithm setting which is enabled by default in the Linux kernel. If the user does two calls to send with 400 bytes each, the first call will result in a new SKB containing the data being added to the TCP output queue. On the second call to send it checks if there is unsent data in the queue, and if there is free space in the newest SKB (which we know it is in this example), the new data will be appended to the newest SKB in the queue.

By disabling Nagle's Algorithm the kernel will not delay sending the data, but instead send the SKB with the new data segment as soon as possible, as long as the send window allows it. Therefore two consecutive calls to send with a small amount of data will result in two packets to be transmitted.

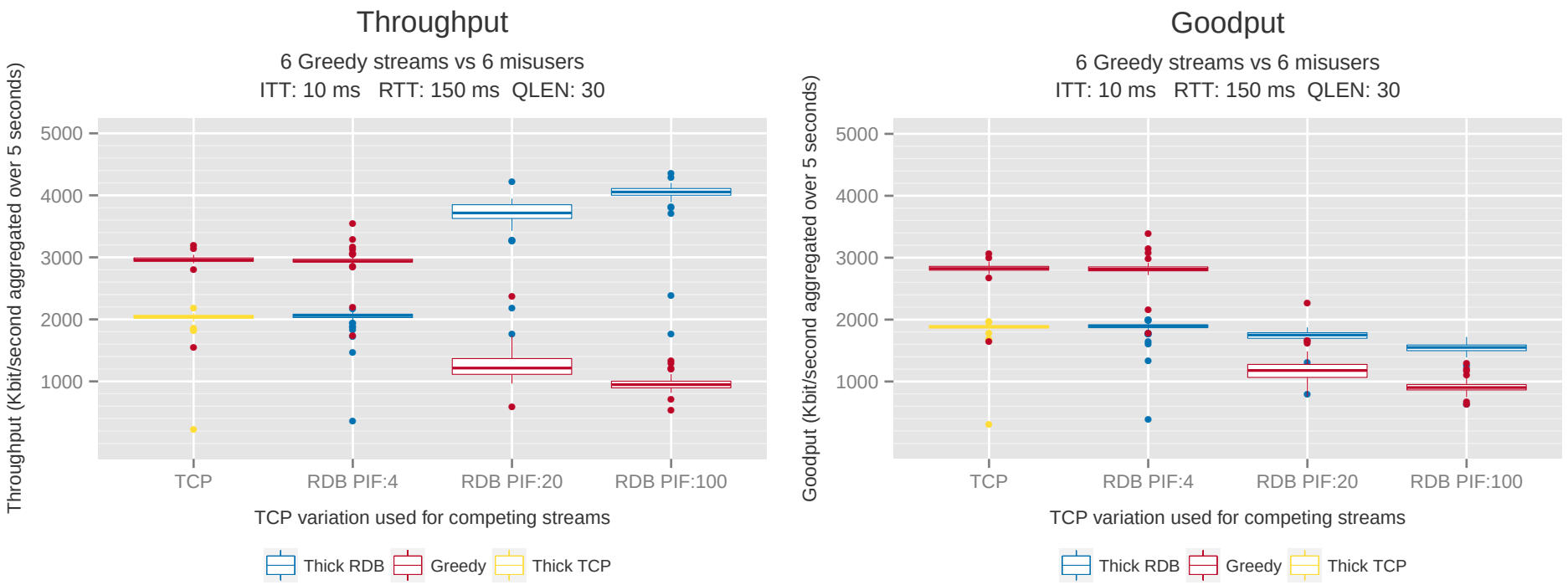
3.2.2.1 Tests with senders abusing and misusing RDB

We ran tests to verify how greedy RDB streams will affect competing greedy streams. We have named these RDB streams *Thick* to indicate that these streams are not requesting a maximum amount of bandwidth as traditional greedy streams. With an ITT of 10 ms and payload of 400 bytes they are "only" requesting 40 KB/s ($40000 = 100 * 400$) which is more than the thin-stream applications in table 2.1, but they would still be application limited in our testbed if there were only a few simultaneous streams.

Subfigure 3.3.(a) shows the result of 6 greedy TCP streams competing with 6 RDB streams. The RDB streams do calls to send with 400 bytes every 10 ms, which gives room to bundle the data of two previously sent packets for each packet sent. Each packet transferred on the wire will contain 1200 bytes, where 800 of them are redundant.

When the RDB streams are limited by maximum three PIFs, they end up not bundling any data, and as a result, get the same throughput as the regular TCP streams. As soon as the limit is raised, the RDB streams get substantially better throughput than the greedy streams.

If the RDB streams in subfigure 3.3.(a) lose two consecutive packets, the data of the first and second lost packet is still received in the packet coming next, which contains all the lost data as well as some new data. Neither the receiver, nor the sender, will know that the data was lost, and the result is that the senders will not back off as the other streams will, due to detecting loss. It is clear that the RDB streams get an unfair advantage by not reacting to loss events.



(a) Throughput

(b) Goodput

PIF (-1) is the threshold for how many packets in flight are allowed when data is bundled, i.e., "PIF:4" means a maximum of three PIFs.

Figure 3.3: The aggregated throughput for greedy streams competing with thick TCP and RDB streams

Table 3.3

Test	Name	Type	Streams	Cong	RTT	ITT	Payload
1	TCP	Thick TCP	6	Cubic	150	10	400
1	TCP	Greedy	6	Cubic	150	NA	NA
2	RDB PIF:4	Thick RDB	6	Cubic	150	10	400
2	RDB PIF:4	Greedy	6	Cubic	150	NA	NA
3	RDB PIF:20	Thick RDB	6	Cubic	150	10	400
3	RDB PIF:20	Greedy	6	Cubic	150	NA	NA
4	RDB PIF:100	Thick RDB	6	Cubic	150	10	400
4	RDB PIF:100	Greedy	6	Cubic	150	NA	NA

Table 3.3: Greedy vs RDB misuser test setup

3.3 PIFs as a thin-stream indicator

As mentioned in section 2.6, the TCP modifications, LT and mFR, are only active on thin streams, assured by the function `tcp_stream_is_thin` in code listing 2.1.

The limit of maximum three PIFs, which is hardcoded in the function `tcp_stream_is_thin`, is justified because with three or fewer PIFs the stream will not receive the three required dupACKs to trigger a fast retransmit within one RTT.

For testing if a stream will have problems triggering a fast retransmit, it works fairly well, but that it also the limited scope it tests for. Using a *static packets in flight limit* **(SPIFL)** as a metric for classifying thin streams in general, is however, to some extent flawed. The main reason is that the PIFs of a flow is highly dependent on the RTT. Therefore, two application limited streams that send the exact same amount of data, may be classified differently, even though their strict latency requirements are the same. In a network with latencies as low as a few milliseconds, the PIFs limit would allow streams, that would be considered greedy in a network with higher latencies, to use the thin-stream modifications. For this reason, the hardcoded SPIFL must be set very low, to minimize the potential for non-thin streams to use the mechanisms.

While many of the applications listed in table 2.1 will frequently have less than four PIFs on a connection with 70 ms RTT, [2], some of them will have more. These streams will fall outside the limited scope of current test, while still, to some degree, suffering from the same problems.

[2]The median RTT of the connections measured on Google's test servers in 2010 is 70 ms (*Dukkipati*, *Refice*, et al. [2010, a36])

3.4 Congestion control: A cause of reduced latencies

A major issue for time-dependent, application limited TCP streams, is that the congestion avoidance technique of drastically reducing the CWND is very harmful to the latency. It is not only an issue for thin streams, but also with applications such as video streaming (*Tan* and *Zakhor* [1999, a37]).

In scenarios where a thin stream is competing with greedy streams, the greedy streams are to blame for the reduced latencies of the thin streams. This is because of how the CC of the greedy streams work, where it cause a repeated filling and draining of the queues on the bottleneck node in the network path (*Stewart* et al. [2011, a9]). By continually trying to utilize any available capacity on the path, by increasing the send rate until losses occur, they ensure that other streams that share a link to the bottleneck node also lose packets.

This might be considered fair practice when the competing traffic *a)* has the the same behavior, and *b)* when the competing traffic does not care about per–packet latency. But for interactive applications producing thin streams, the consequences are drastic, and unfair. Even when the thin streams are only requesting a small share of the total link capacity, they are forced to reduce their send rate.

Reducing the negative effect the CC has on such network streams has been one of the driving forces behind equation based CCs such as TFRC and binomial CC. By growing in a less aggressive manner compared to slow-start, they argue that it is fair to reduce the send rate in a slower manner.

What is the problem, exactly?

The most obvious problem is that the thin streams, who are only transmitting a very limited amount of data to begin with, are forced to lower their send rate due to the aggressive behavior of greedy streams. Due to the lowered send rate, some of the data must be queued in the TCP output queue for a certain time, causing increased latencies.

The function in the CC that controls how the CWND grows is implemented in the entry point `cong_avoid` in the Linux CC framework (code listing B.13 line 15). The implementation for TCP New Reno can be seen in Code listing B.1, but the logic is similar for other CCs like TCP Cubic. The function tests if the connection is in slow-start (Code listing B.1 line 16), and if so, grows the CWND accordingly. If not in slow-start mode, meaning congestion avoidance mode, the CWND grows in a slower manner.

Some CC implementations, such as TCP New Reno and TCP Cubic also perform a test at the beginning, by calling `tcp_is_cwnd_limited` (Code listing B.1 line 12). This seemingly harmless test is what causes problems for thin streams.

The function `tcp_is_cwnd_limited` tests if the connection's send rate is currently limited by the CWND. If the connection does not utilize the available CWND, it presumably does not need a bigger CWND. While this is reasonable for a greedy stream, as it will prevent it stream from building up CWND that the network cannot support, it causes issues for thin streams. When a thin stream loses

a packet, the AIMD algorithm will reduce the CWND drastically, causing data to be queued in the TCP output queue instead of being sent immediately.

One might wonder if it really is such a big problem, as the CWND would eventually grow back for the thin streams, and again allow them to send the data immediately without any queuing delay. While this is true, the lowered send rate increases the latency, causing reduced service quality.

Recent changes to the Linux kernel's TCP

In Linux kernel version 3.16, the function `tcp_is_cwnd_limited` was rewritten, which changes the situation for thin streams. The two commits e114a7 and ca8a22, which we believe can be attributed to the effects we observe, cause increased latencies for thin streams.

The implementation in Linux kernel version 3.15 (code listing B.2 would, at call time, simply test if the number of PIFs was not less than the CWND. The first commit (e114a7) aims to improve the accuracy of when `tcp_is_cwnd_limited` reports a stream as limited by the CWND. An important change is that it moves the $PIF < CWND$ test into the CWND validation mechanism (`tcp_cwnd_validate`) performed in the TCP output engine. Therefore, the CCs no longer test if the stream is network limited at the time it is asked to grow the CWND.

The second commit (ca8a22) tries to fix a problem with the first commit, where the CWND could be larger than necessary. The result is that after the CWND is reduced due to packet loss, the CWND does not grow back to the previous level. This prevents the sender from maintaining the number of PIFs it needs to transmit the data segments without being delayed in the TCP output queue. When the CWND does not grow, more data is buffered in the TCP output queue leading to bigger packets being sent, which effectively enforces a behavior similar to Nagle's Algorithm.

Whether the effect on thin streams is unintended, meaning it can be regarded as a bug, or if this is really how the kernel developers intended it, is yet to be found out. This behavior has been confirmed up to Linux kernel version 3.19-rc7, the latest release at the time of writing. The implementation of `tcp_is_cwnd_limited` in Linux kernel version 3.16 is shown in code listing B.3.

3.5 Summary

In this chapter we have looked at the latency for thin stream, and what causes increased latencies for such streams. We have presented the results from a set of preliminary experiments, that tests thin streams using the mechanisms mFR, LT and ER+TLP against competing greedy traffic. From the results we see how the loss of a packet reduces the latency for many of the following packets due to HOL blocking.

We have gone into details about different issues with the RDBv1 implementation as well as topics on how to detect if a stream is thin. We have described how the traditional CCs for TCP control the CWND can cause increased latencies

for thin streams, as well as how changes to the CC in Linux kernel3.16 further increases the latencies for thin streams.

Chapter 4

RDB prototype v2

As discussed in section 3.2, RDBv1's bundling scheme is considered too aggressive. The increased network data for a thin stream using RDB may be considered a problem, however, compared to greedy streams, the extra load on the network is minimal. However, the problem of RDB hiding loss is strictly necessary to address, as it prevents the TCP engine and the regular CCs from controlling the stream in response to network congestion.

For this reason, we have investigated how to make RDB less aggressive, where our main goal has been to ensure that RDBv2 behaves in a TCP friendly manner towards competing network traffic, while still being effective in reducing the application layer latency for thin streams.

I this chapter, we address the issues described in chapter 3, and present how we tried to solve them in RDBv2. We first look at the thin-stream classification mechanism described in section 3.3, and how it can be improved to more precisely identify thin stream. Further, we go into detail about how to detect the hidden loss in an RDB stream, before we describe the re-implementation of the bundling code and how it differs from the RDBv1 implementation. Next, explain the background for the CC mechanism we have implemented in RDBv2, which we refer to as RDBv2-CC.

4.1 Classifying thin streams

The Linux kernel "defines" a stream as thin by the number of PIFs being less than four (see code listing 2.1), and uses this to decide whether to use the mechanisms `tcp_thin_dupack` and `tcp_thin_linear_timeouts` described in section 2.6.

Using the same test to restrict RDB would certainly make RDBv2 less aggressive than RDBv1, but that would result in no redundant bundling in many circumstances where it could be beneficial, e.g., when the RTT is so high that the PIFs exceed three very easily.

As described in section 3.3, the use of the static PIF value is somewhat flawed, as the underlying network environment is not taken into account. If a PIF limit should make sense, it must be calculated so that an application is treated similarly under different network environments. The SPIFL complied by the function `tcp_stream_is_thin` will be stricter to a stream with a higher RTT, even when the

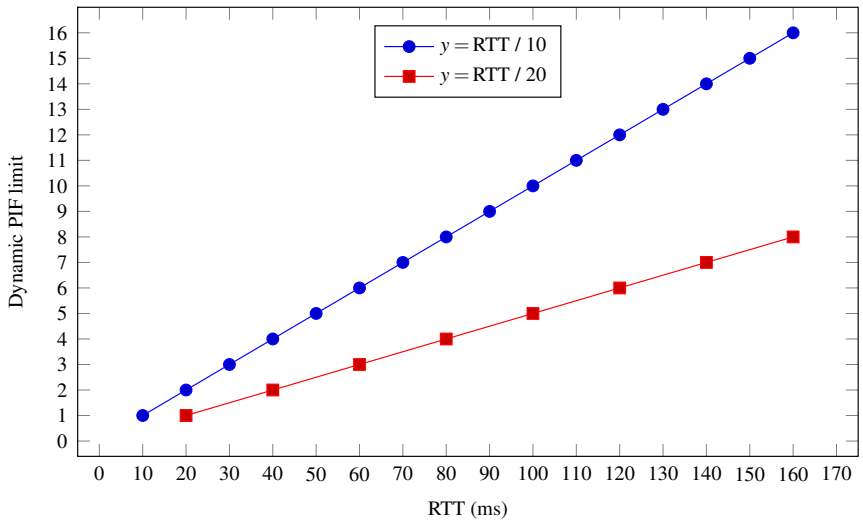


Figure 4.1: The DPIFL with minimum ITTs 10 ms and 20 ms, for RTTs in range 10 – 160

sending rate remains constant. Therefore, the stream's RTT should be taken into account when limiting based on PIFs.

By basing the limitation on the application's sending rate, it is easier to make a distinction between the applications we believe deserve to benefit from RDB, and those who do not.

We have therefore chosen to do this by basing the thin-stream classification on a maximum number of PIFs relative to the RTT. By dynamically calculating the PIF limit as factor of the flow's RTT, we ensure that network streams are treated more equally under differing network environments. This dynamic limit, which we call *dynamic packet in flight limit* **(DPIFL)**, is calculated according to equation 4.1,

$$DPIFL = \frac{RTT_{min}}{ITT_{min}} \tag{4.1}$$

where RTT_{min} is the minimum RTT value observed, and ITT_{min} is a static limit defining the minimum allowed ITT.

Figure 4.1 shows the DPIFL plotted for RTTs ranging from 10 to 160 ms, with minimum ITTs of 10 ms and 20 ms. The ratio between DPIFL-10 and DPIFL-20 is 1 : 2, such that DPIFL-10 allows for the double number of PIFs compared to DPIFL-20 on the same RTT.

4.2 Loss detection

The fact that RDB hides loss events, as described in section 3.2.1, causes it to not react well to losses in many situations. It gives the streams an unfair advantage, which goes against the TCP-friendliness concept described in section 2.5.3. We have therefore investigated how to detect loss events that RDB hides from the TCP engine.

When the TCP data sender receives a dupACK, it must consider if it was caused by reordering or by loss. With RDB, the potentially lost or reordered data is included with the next packet. Therefore, dupACKs are not sent when sporadic loss of only one packet occurs, as explained in section 3.2.1. The probability of receiving dupACKs depend solely on how many (already sent) data segments are being bundled with each packet. If each RDB packet includes the bundled data of four previously sent packets, the loss of five consecutive packets is required for a gap in the data to appear on the receiver side, resulting in a dupACK being sent.

We have investigated the following techniques for detecting packet loss in an RDB stream:

- **Multiple packets ACKed**
  When an ACK covers multiple SKBs, it is a good indicator that a packet has been lost. This test alone cannot distinguish between a lost data packet and a lost ACK. If reordering occurs, this technique may also fail. However, for thin streams, the high ITT itself reduces the chances of packets arriving out-of-order.

- **TCP time stamps**
  We investigated if TCP timestamps could be used to improve the loss detection accuracy, similarly to how the Eifel detection algorithm, described in section 2.4.5.2, distinguishes between reordering and loss.

- **DSACK**
  The DSACK extension to SACK, enables the receiver to accurately inform the sender of duplicate received data segments. We have looked at how this mechanism behaves in a RDB scenario, and how this could be used for loss detection.

In the next sections, we will go into detail about these two techniques for detecting loss.

4.2.0.2 ACKs covering multiple segments indicating loss

One idea for detecting lost packets is to look at the received ACKs. If a sender can expect to receive one ACK for every data packet sent, it can presume that any deviations to this is caused by data packets, or ACKs on the reverse path, being lost. If an incoming ACK acknowledges multiple SKBs in the TCP output queue, it must be caused by:

- a data packet being lost, resulting in all or some of the redundant data in the next packet effectively not being redundant after all.
- an ACK on the reverse path being lost, causing the ACK on the next packet to advance snd_una[1] by two segments.

There are only two flaws in this technique. One being packet reordering, which can cause the first ACK to be a dupACK, and the second ACK to acknowledge two segments. In that scenario, it would be incorrect to consider the second ACK, that acknowledges two packets, as an indication of loss. The other problem is caused by lost ACK packets. Distinguishing between loss of data packets, and loss of ACK packets on the reverse path can be difficult, and with this technique alone, it is not possible.

Delayed ACKs and RDB

A regular TCP stream must consider that TCP delayed acknowledgment may be enabled on the receiver. Receiving ACKs that span (acknowledge) multiple segments, is therefore a part of the normal workflow, and should not be treated as a possible loss signal. Due to this, the loss detection technique described in section 4.2.0.2 cannot be used for regular TCP.

However, delayed ACKs are used only when data is received in sequence, as seen on line 3 in code listing 2.3. When RDB bundles previously sent data, the sequence number does not equal the expected sequence number, and TCP will perform a quick ack by calling `tcp_enter_quickack_mode` (see code listing 2.3 line 15 and 22). Therefore we know that when sending an RDB packet, an ACK will be sent for each data packet delivered to the receiver.

4.2.0.3 TCP Timestamps

Similar to how the Eifel detection algorithm solves the *retransmission ambiguity problem*, we looked at how TCP timestamps can be used to identify lost packets in an RDB stream.

Figure 4.2 shows a scenario where an RDB stream bundles one older packet with each data packet. When packet *B* is lost, the next ACK sent in reply, acknowledges segment *C* using *C*'s `TSval` in the *echo-reply* field. Due to the requirement that the *echo-reply* field of an ACK, always must reflect the earliest data segment that *this* ACK acknowledges, the *echo-reply* field in dupACKs will never reflect the packet that triggered it. This is because dupACKs do not acknowledge any new data.

When a dupACKs is received, with the *echo-reply* field reflecting segment *C*, we can speculate that the previously missing ACK was simply caused by a reordering, and not by a lost packet. However, if there is more outstanding data that contains segments newer than *C*, it is an ambiguous situation, as the *echo-reply* does not tell you if the incomming packet had old data (before *C*), or contained

[1] snd_una is a variable in the TCP socket struct used to keep track of the greatest un-ACKed sequence number.

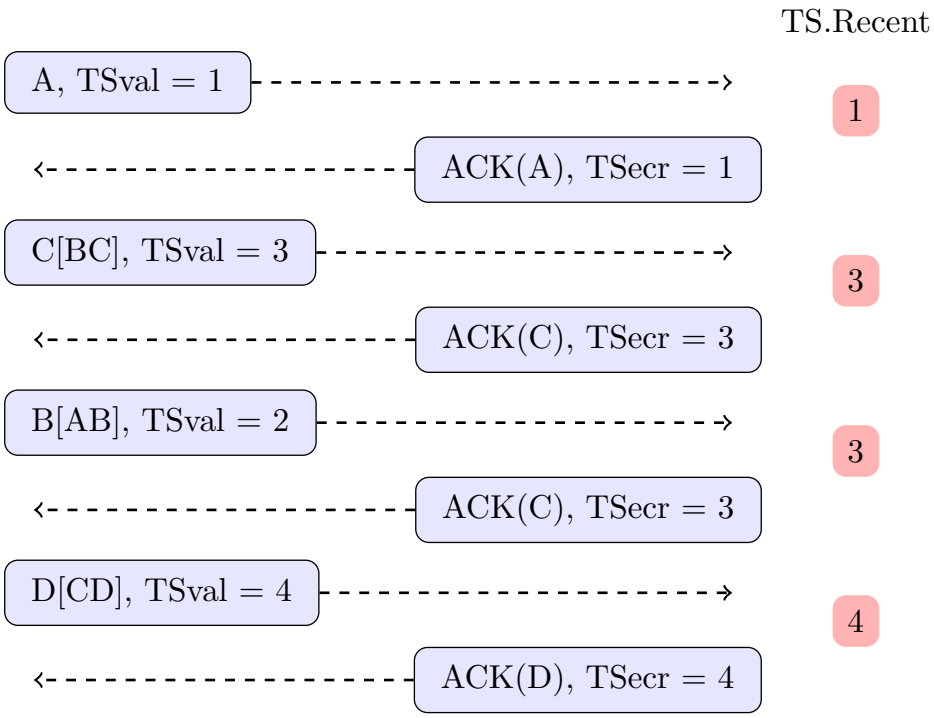


Figure 4.2: Example of TCP timestamps with TCP delayed acknowledgment (From *RFC7323*)

data newer than the highest in-sequence segment. Had the *echo-reply* of dupACKs reflected the timestamp of the incomming packet that triggered it, it would be possible to distinguish between the two cases.

4.2.0.4 DSACK

As described in section 2.4.1, the DSACK extension to SACK enables the receiver to accurately inform the sender of duplicate received data segments. The consequence is that when DSACK is enabled on an RDB stream, the redundant data bundled with newer packets causes the receiver to populate the SACK option field with the range of already received data. This can be used to identify lost packets.

Figure 4.3 shows an RDB stream that bundles one previous segment with each data packet. As we can see, each ACK's SACK field is populated with the DSACK range of the redundant segment.

Figure 4.4 illustrates how it looks when reordering occurs. Due to reordering of one packet, the third data packet arriving on the receiver side contains no old (already received) data, and consequently, the ACK sent in return does not contain any SACK information. This will only happen if reordering has occured, or if one packet was lost.

If an ACK is lost, the data sender will notice that an ACK was skipped, but due to the DSACK range in the next ACK, it can conclude that the data had already

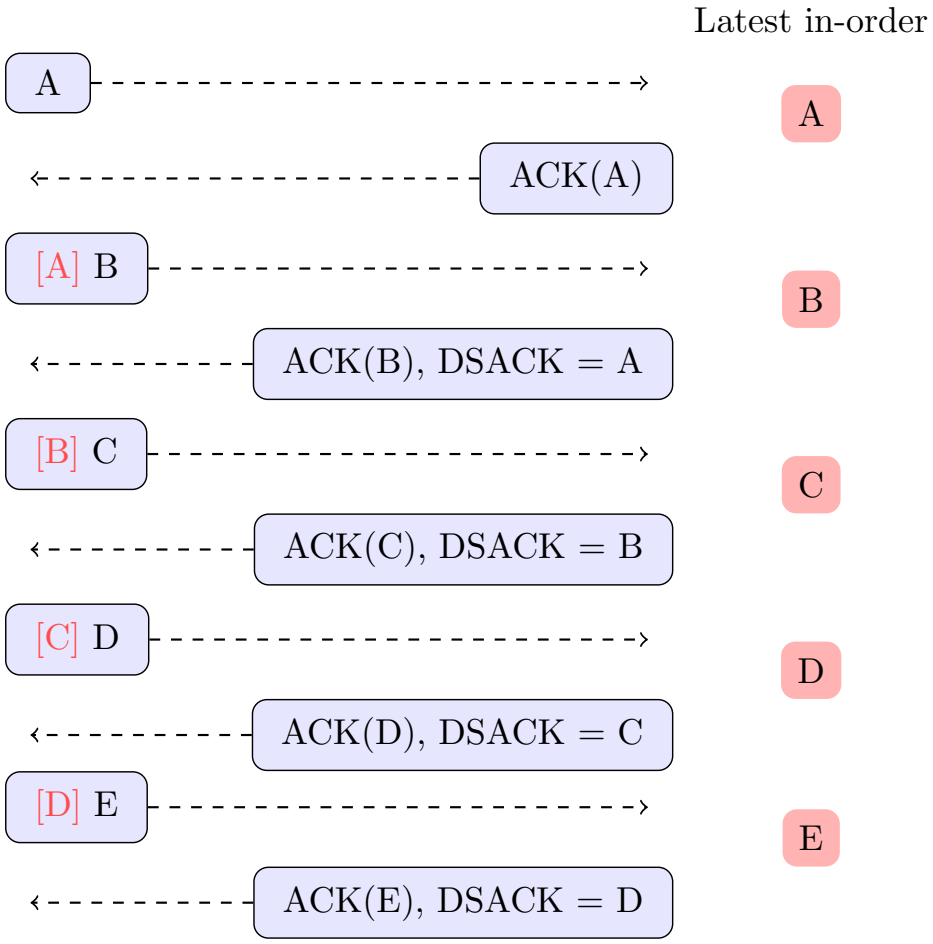


Figure 4.3: Example of RDB stream with SACK option enabled.

been received, and hence, it must be the ACK that was lost.

The SACK option, along with the DSACK extension, will help us solve the ambigous case that the TCP timestamps could not. By providing a DSACK range when the packet that triggered the dupACK contains old data, and a regular SACK range when the data is newer than the highest in-sequence segment, we can unambigously identify which packet triggered the dupACK in the case of reordering.

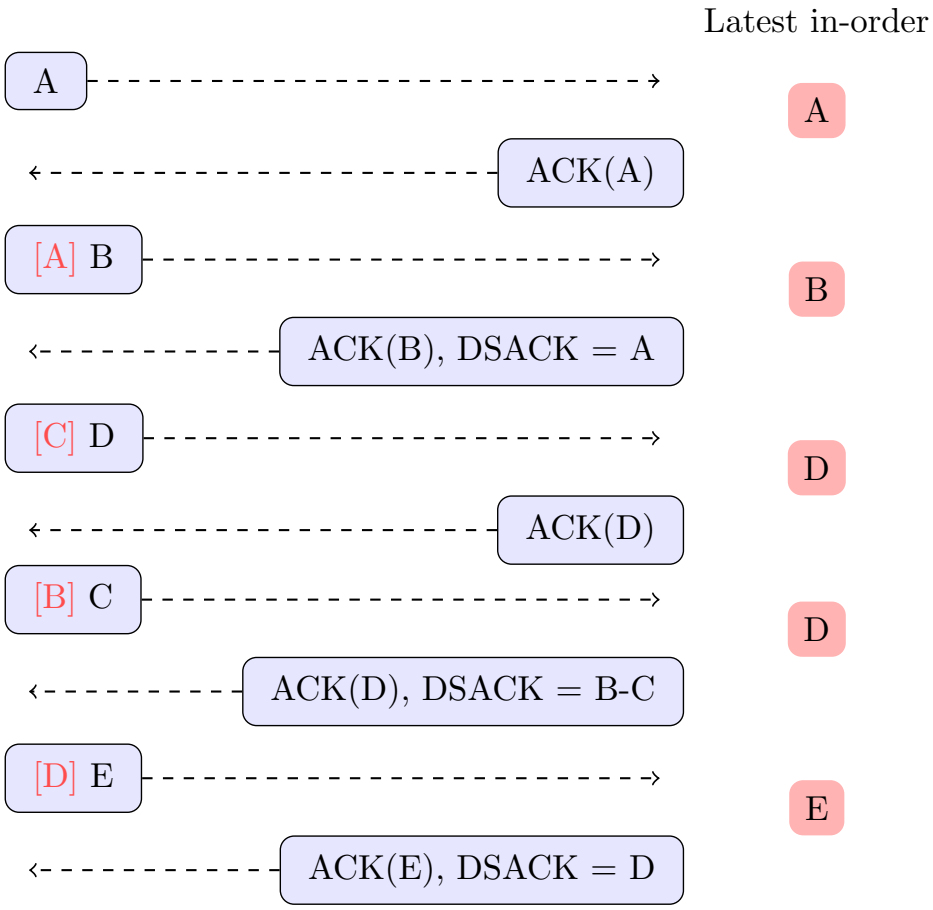


Figure 4.4: Example of how the DSACK is used on an RDB stream with packet reordering.

4.3 Implementing redundant bundling

Re-implementing the bundling code was necessary, because the initial implementation (RDBv1) turned out to be more intrusive to the core of the Linux kernel's TCP engine than strictly necessary.

In a response on the Linux net-dev mailing list, Ilpo Järvinen suggested a different approach than what was used in RDBv1. Instead of modifying the payload of the SKBs in the TCP output queue when receiving data from user space, he suggested to create the SKBs containing redundant data on-the-fly at send time (email D.1). This approach would avoid any modifications to the data in the TCP output queue, and allow cleaner entry points into the codebase.

The part of RDBv2 that implements bundling is a re-implementation of the functionality implemented in RDBv1. While, functionally, there is little difference between these implementations, RDBv2 has been structured differently, and organized into separate files which makes it easier to read and understand.

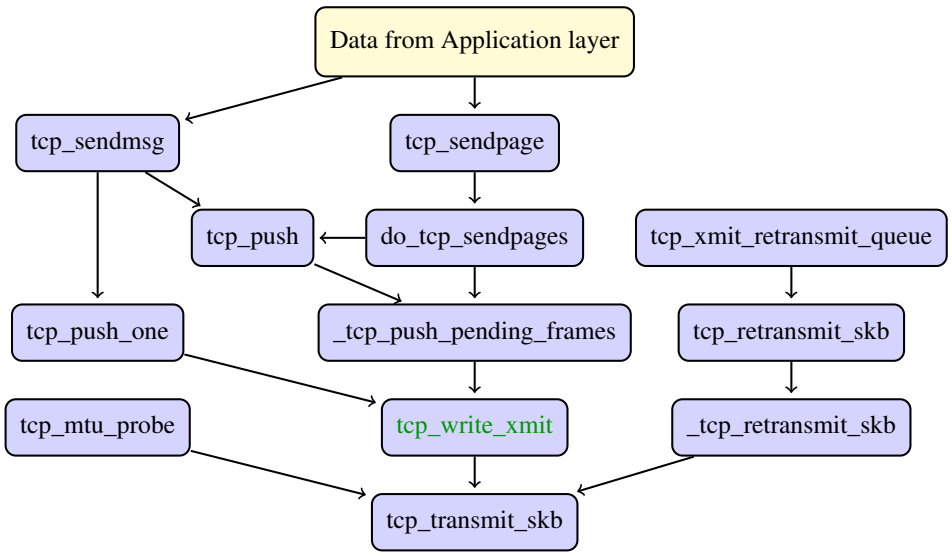


Figure 4.5: Call graph for TCP output engine, where the changes made for RDBv2 are marked in green.

For convenience, the code listings for the implementation is placed in the appendix B, and is referred to in the descriptions of each function.

4.3.1 Entry point for sending custom SKBs

The code in RDBv2 responsible for modifying the SKBs, is modeled after the suggestion by Ilpo Järvinen (see email D.1). Instead of modifying the SKBs directly in the TCP output queue, RDBv2 creates a new SKBs on the fly, and copies redundant data from the SKBs in the TCP output queue into the linear page buffer of the new SKB.

All the call-paths in the program flow for sending new data (TCP output engine, `tcp_output.c`) ends up in the function `tcp_write_xmit` (see figure 4.5).

The function `tcp_write_xmit` (see code listing B.4) is called to write packets with unsent data to the network. It will keep sending packets from the TCP output queue as long as they fit into the send window, and the argument for the `push_one` parameter is 0.

In the function `tcp_write_xmit`, we have added a call to a new entry point (`pkt_send`) in the Linux CC framework (see code listing B.13 line 25).

As shown in code listing B.4, the call to the new entry point is added just before the call to `tcp_transmit_skb` (Line 13). If the return value of the `pkt_send` function is zero, the packet has been successfully sent, and the default call to `tcp_transmit_skb` is skipped (Line 19). If the return value is not zero, the code proceeds to execute the standard call to `tcp_transmit_skb` (Line 24).

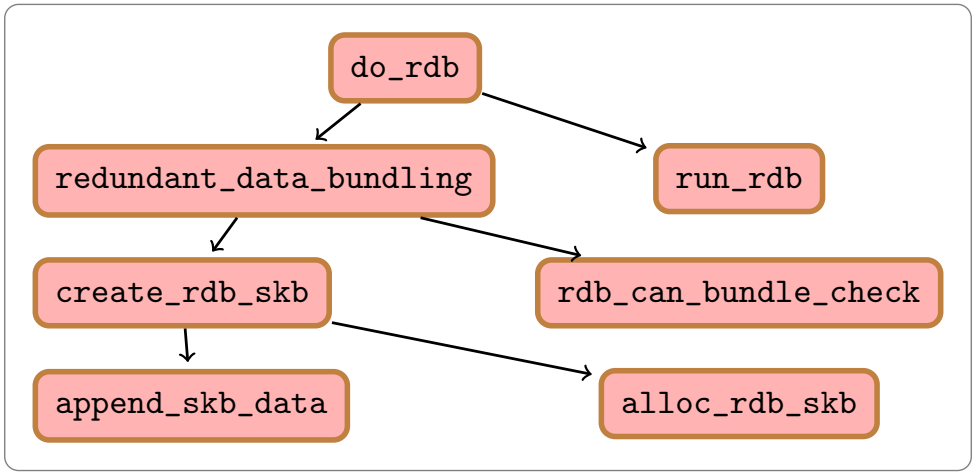


Figure 4.6: Callgraph for the code in RDBv2 that performs the redundant bundling.

4.3.2 Performing redundant data bundling

The code that tests if an RDB packet may be created, as well as creates and sends the packet, is implemented in `rdb_skb.c`. The function `do_rdb` in `rdb_cc.c` is called through the entry point `pkt_send`, as shown in figure 4.10.

We show shortened versions of the most relevant functions for performing the bundling, and explain their main tasks:

`do_rdb()`

This function (see code listing B.5) is called by the `pkt_send` entry point, and does the following:

- Calls `run_rdb` (Code listing B.6) which tests if the RDB socket option is set, and that the stream is *thin* by calling `tcp_stream_is_thin_dpif`.
- Calls `redundant_data_bundling` (Code listing B.9) which tests if it is possible to bundle. If so, it creates a new SKB with redundant data.
- Calls `tcp_transmit_skb` to transmit the RDB packet..

`run_rdb()`

This function performs the initial test to see if the stream is currently eligible for redundant data bundling. It will first test if the `TCP_THIN_RDB` socket option has been set (see line 2 in code listing B.6). Next it will test wether to use DPIFL or SPIFL to classify the stream is *thin* or not.

If DPIFL is enabled, it will use the function `tcp_stream_is_thin_dpif` (code listing B.7), which implements the DPIFL test described in section 4.1.

If DPIFL is not enabled, it will use the function `tcp_stream_is_thin_spif` (code listing B.8), which is is a modified version of `tcp_stream_is_thin` (code listing 2.1). To easily perform tests with different SPIFL values, this function reads the PIF limit from a sysctl variable instead of using a hardcoded limit. The second difference is that we have removed the call to `tcp_in_initial_slow-start`, which restricts the thin-stream mechanisms to streams that have experienced packet loss. This is because we want to bundle data even before loss has occured.

`redundant_data_bundling()`

The function `redundant_data_bundling` (Code listing B.9) tests if it is possible to bundle any redundant data, and if so, returns a new SKB containing redundant data followed by the new data to be transmitted.

- First, it calls `rdb_can_bundle_check` (Code listing B.10) which traverses the TCP output queue, and returns the first SKB (with the oldest data) that can be bundled.
- If a valid SKB is returned by `rdb_can_bundle_check`, it calls `create_rdb_-skb` which creates the new SKB, and returns it. The resulting SKB will contain some new data, as well as the data from one or more older SKB, such as depticted in figure 2.10.

`rdb_can_bundle_check()`

`rdb_can_bundle_check` (Code listing B.10) goes through the TCP output queue to find the first (oldest) SKB containing data that can be bundled. Starting at the SKB that the TCP output engine tries to send, it traverses the queue in reverse order and tests if the data of each of the SKBs can be included in the new SKB. If there is room left for parts of the data in an SKB, it will save the offset into the data buffer of that SKB, which is passed into the function `create_rdb_skb` that creates the SKB and copies the data.

To control the amount of bundling in each SKB, this function also uses two sysctl variables, `sysctl_tcp_rdb_max_bundle_bytes` and `sysctl_tcp_rdb_-max_bundle_skbs`. These variables can optionally be set before a test, which will be used for all the RDB streams in the test. If any of these variables are not zero, they will limit the maximum number of bundled bytes per RDB packet, or the maximum number of SKBs to bundle from.

4.3.3 Bundling on retransmission

To measure the effect of bundling on retransmissions, a separate set of tests would be required, so that the results of bundling on retransmissions could be compared to the tests without bundling on retransmissions. To reduce the number of experiments, we decided to only test bundling when transmitting new data.

The RDBv2 code presented here does not implement the `pkt_send` function hook on retransmission, however, the changes required to do so is simple, and very similar to the code on line 13 in code listing B.4.

4.4 RDB Congestion control background

In addition to the functionality required to bundle data, the RDBv2 implementation contains a CC that aims at solving a range of issues. Dealing with the issue of RDB hiding loss events (described in section 3.2.1) is strictly necessary to make it TCP friendly. Also, preventing abuse, by implementing functionality to limit when bundling may occur, is a strict requirement to ensure TCP-friendliness. Solving these main tasks could best be done by creating a new CC module.

4.4.1 TFRC-like congestion control

TFRC as specified in (*RFC5348*) requires the receiver side to calculate the loss rates and report this to the sender side in feedback packets. One advantage to this is robustness to loss of the feedback packets, as this would not influence the loss rate calculations.

An important benefit with the initial RDB implementation, RDBv1, is that only sender side modifications are required. By introducing CC functionality that requires receiver side modifications, we would not be able to keep the TCP compatibility which is one of the main goals we have set for RDBv2.

RFC5348 also specifies sender-based variants, where the loss rate calculation is moved to the sender side, which is what we have chosen to base our implementation.

The goal of TFRC is to provide a CC that has less variation of the throughput compared to TCP. Where TCP reacts drastically to loss by either halving the CWND, or in the case of an RTO, set it to one, TFRC seeks to avoid these drastic reductions to the send rate by calculating the average loss over a certain period of time. In the case of just a single lost packet, the CWND is not reduced as much as for TCP. While TCP will not reduce the CWND for multiple lost packets within the same window, it will not losses in different windows any differently.

TFRC on the other hand, divides the sequence space into loss intervals, which makes a loss history that is used to calculate the allowed send rate. In addition to packet losses, congestion events, such as ECN marked packets, can be used to build the loss history.

From the packet history, a loss event rate p is calculated, which in turn determines the speed at which TFRC responds to congestion.

4.4.1.1 Loss History

Where TCP sequence numbers are based on the number of bytes being sent as payload, TFRC packets are given a sequence number which is incremented by one for each packet. The length of a loss interval is specified in *RFC5348* as:

“ If a loss interval, A, is determined to have started with packet sequence number S_A and the next loss interval, B, started with packet sequence number S_B, then the number of packets in loss interval A is given by (S_B - S_A).”

This approach enables the sender to calculate how serious the congestion is by looking at how many packets have been lost over time.

Like TCP, TFRC will treat multiple lost packets within the same send window differently. If a lost packet is within the same RTT as the lost packet that started the current interval, it will include it in the current interval instead of creating a new one.

The number of loss intervals contained in the loss history affects how quickly the loss event rate is adjusted when the loss rate changes. The more loss intervals used, the slower the loss event rate will adjust. For TFRC, it is specifies that eight loss intervals should be used.

4.4.1.2 Calculating average loss interval

A sequence of weights are used to ensure that the loss event rate changes smoothly. From *RFC5348*, the weights are calculated by pseudo code 4.1, such that the first four (of eight) weights are 1.0 (meaning no change), and the next weights reduce the significance of the values to be weighted more and more the older they are.

The average loss interval is calculated according to pseudo code 4.2, where the weights are applied to the loss intervals. Two calculations of the average loss interval are made, with and without the the current loss interval, i.e., the loss interval that is currently open. The maximum of the two calculated values is used as the mean value, such that the current loss interval will only be used if it is large enough to increase the average loss interval.

Pseudo Code

```
Weights w_0 to w_(n-1) are calculated as:

  If (i < n/2) {
      w_i = 1;
  } Else {
      w_i = 2 * (n-i)/(n+2);
  }

Thus, if n=8, the values of w_0 to w_7 are:

  1.0, 1.0, 1.0, 1.0, 0.8, 0.6, 0.4, 0.2
```

Code Listing 4.1: TFRC pseudocode for calculating the weighting for the average loss intervals (*RFC5348*)

Pseudo Code

```
  I_tot0 = 0;
  I_tot1 = 0;
  W_tot = 0;
  for (i = 0 to k-1) {
      I_tot0 = I_tot0 + (I_i * w_i);
      W_tot = W_tot + w_i;
  }
  for (i = 1 to k) {
      I_tot1 = I_tot1 + (I_i * w_(i-1));
  }
  I_tot = max(I_tot0, I_tot1);
  I_mean = I_tot/W_tot;

The loss event rate, p is simply:

  p = 1 / I_mean;
```

Code Listing 4.2: TFRC pseudocode for calculating average loss interval (*RFC5348*)

Pseudo Code

```
  I_tot0 = 0;
  I_tot1 = 0;
  W_tot = 0;
  for (i = 0 to n-1) {
    I_tot0 = I_tot0 + (I_i * w_i);
    W_tot = W_tot + w_i;
  }
  for (i = 1 to n) {
    I_tot1 = I_tot1 + (I_i * w_(i-1));
  }
  If the current loss interval I_0 is "short"
    then I_tot = I_tot1;
    else I_tot = max(I_tot0, I_tot1);
  I_mean = I_tot/W_tot;

The loss event rate, p:

  p = 1 / I_mean;
```

Code Listing 4.3: TFRC-SP pseudocode for calculating average loss interval (*RFC4828*)

4.4.1.3 Calculating the send rate

As the unit of the calculated send rate from equation 2.2 is bytes/s, and the sender is limited by the CWND, the send rate is converted into a CWND value by equa-

tion 4.2.

$$RTT_{sec} = \frac{10^6}{srtt_{\mu s}} \qquad X_{RTT} = \frac{X_{sec}}{RTT_{sec}}$$
$$CWND = \frac{X_{RTT}}{psize} = \frac{X_{sec} * srtt_{\mu s}}{10^6 * psize} \tag{4.2}$$

We first divide the send rate in seconds (X_{sec}) by the number of RTTs in a second (RTT_{sec}) to get the send rate per RTT (X_{RTT}). We then divide X_{RTT} by the the size of the packets (*psize*) to get the corresponding CWND value. For the *psize* we use the same value as used for packet payload size in the throughput calculation (Equation 2.2).

The variable b in equation 2.2 denotes the maximum number of packets that can be acknowledged by a single ACK. Due to TCP delayed acknowledgment, this will normally be 2, but *RFC5348* recommends using 1, because TCP is allowed to send an ACK for every packet. Considering that RDB effectively disables TCP delayed acknowledgment, we are safe use to 1 for the calculations.

4.4.1.4 TFRC Small-Packet variant

RFC4828 defines *TCP-Friendly Rate Control: The Small-Packet (SP) Variant* **(TFRC-SP)**, a modification of TFRC for applications that send small packets, where the design goal is to achieve the same bandwidth as a TCP stream sending packets of size MSS. *RFC4828* specifies that a minimum ITT of 10 ms should be enforced to prevent a stream from sending too many small packets. This limit is based on the requirement of VoIP applications, that do not send packets more often than 10 ms. Also, this limit is justified by the lack of applications used in in the current Internet that require to send small packets more often.

To accommodate applications sending time-dependent data, TFRC-SP proposes a set of adjustments that will help streams with thin-stream characteristics.

Loss Event Rate

TFRC competes with TCP by counting the loss event rate, where one loss event is one or more lost packets within a window of data. A thin stream that sends multiple smaller packets with the same bit rate as standard TCP sending fewer MSS sized packets, can no longer achieve fairness with TCP by counting the loss event rate as defined in *RFC5348*. The loss measurement for a thin stream must be able to detect when a stream consistently loses multiple packets within an RTT. Therefore, TFRC-SP requires that for loss intervals that are longer than two RTTs, one loss event is counted. For shorter intervals, the lost or marked packets are accounted for by calculating the interval length as N/K where N is the number of packets in the interval, and K is the number of loss events in the interval.

Taking packet headers into account

By sending multiple smaller packets instead of fewer larger ones, a stream will cause extra load on the nodes along the network path. To account for this, TFRC-SP proposes to include the packet header size when calculating the throughput, as shown in equation 4.3.

$$X := X * s\_true/(s\_true + H), \tag{4.3}$$

X denotes the send rate, s_true is the average packet payload size, and H is the expected per packet header size. Unless the exact header size is easily available, a default size of 40 bytes is used for both IPv4 and IPv6, which is justified as a rough fairness is sufficient.

4.4.1.5 TFRC-SP simulations

Before implementing the TFRC-SP concept into the CC of RDBv2, we decided to do some simple simulations to get an idea of how (well) it works. We made a simple python script that implements the throughput calculation from equation 2.2 (as seen in code listing B.11), as well as the calculation of the average loss interval from pseudo code 4.3 (as seen in code listing B.12).

The simulations were performed with 20 individual streams, with the loss rates 0.1%, 0.2%, 0.5%, 1%, 2%, and 5%. For comparison, we ran testbed experiments with uniform loss induced with `netem`, with the same loss rates as the simulations. We ran the testbed experiments for 5 minutes, with the same number of simultaneous streams, using an RTT of 150 ms, which is also used in the experiments detailed in section 5.4. To generate the thin streams, we used `Streamzero` with an ITT of 50 ms, which should give each stream around 3 PIFs (150/50).

How the testbed experiments are setup and run is explained in detail in chapter 5.

Simulation Results

From the results of the testbed experiments shown in figure 4.7, we see a clear trend where the CWND stabilizes at a low value around 2-4 as the loss rates grow. This illustrates the problem described in section 3.4 well, where the CWND does not grow more than what the requested send rate requires.

The results from the simulations in figure 4.8 show a very different picture compared to the testbed experiments of TCP. While the simulation with the highest loss rate of 5% shows that the streams are granted a low CWND, equal to the CWND values of the testbed experiments, the simulations with lower loss rates show that the CWND steadily increases as loss rates decrease.

This is exactly what we want for the thin streams - a heavy reduction of the CWND when high loss rates are encountered, and a less aggressive reduction on lower loss rates.

Figure 4.7: Congestion window growth for 20 thin streams with TCP Cubic using default kernel settings

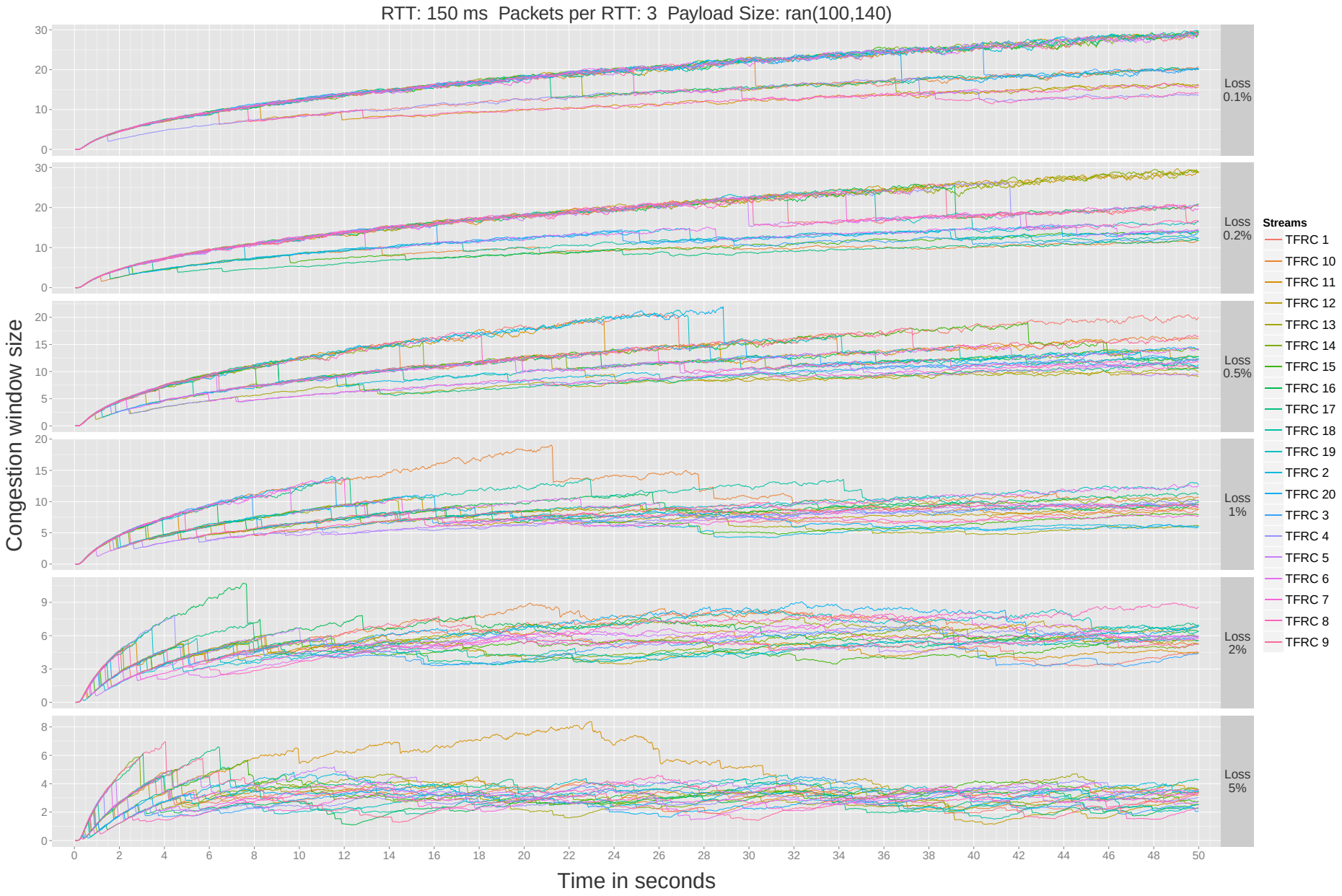


Figure 4.8: Congestion window simulation for 20 thin streams using TFRC-SP

4.4.2 Main tasks of the RDB congestion control

The RDBv2-CC aims to solve the following tasks:

- **Deal with the "hidden loss events" issue** caused by RDB described in section 3.2.1. We handle this issue by using the technique described in section 4.2.0.2 for detecting the hidden loss. The detected loss events are recorded in the TFRC loss history described in section 4.4.1.1, and used in the loss event rate calculation described in section 4.4.1.2.

- **Alleviate the issue of CWND growth for thin streams** described in section 3.4. This is solved by using the implementation of the function `tcp_is_cwnd_limited` from Linux kernel version 3.15, which is less strict than the version implemented in version 3.16.

- **Alleviate the issue of aggressive CWND reduction harming thin streams** described in section 3.4. We try to solve this by adjusting the CWND based on the TFRC throughput calculation designed to produce a more steady throughput rate. On every loss event, the CWND is adjusted according to the send rate calculated by equation 4.2.

- **Ensure that users do not have incentives for misusing RDB** by choosing RDB instead of standard TCP, for the purpose of increasing throughput at the expense of competing traffic. To make RDBv2 less attractive for throughput hungry streams, the CWND growth rate is reduced by lowering the slow-start threshold compared to CCs like TCP Cubic and TCP New Reno.

- **Prevent abuse of RDB** as described in section 3.2.2. This issue is dealt with by limiting bundling to a maximum number of PIFs similarly to the limitation imposed for the mechanisms mFR and LT. However, instead of a SPIFL of maximum 3 used by `tcp_stream_is_thin` (code listing 2.1), a DPIFL is used, which is a PIF limit dynamically calculated using equation 4.1. While the RDBv2 implementation can set the ITT_{min} manually, the default value is set to 10 ms, which is the minimum allowed ITT for a stream using TFRC-SP (*RFC4828*).

4.5 RDB-CC Implementation

RDBv2 has been implemented as a pluggable congestion control module. As the implementation spans multiple files, they have been placed in a new sub-directory `net/ipv4/rdb/`. The implementation of TFRC specific functionality has been placed in `net/ipv4/rdb/lib/`.

4.5.1 Kernel Module

RDBv1 was for the most part implemented in the files `tcp.c`, `tcp_input.c` and `tcp_output.c` in the Linux kernel source tree. After starting to implement RDBv2, we eventually found the need to modify parts of the CC. We decided to

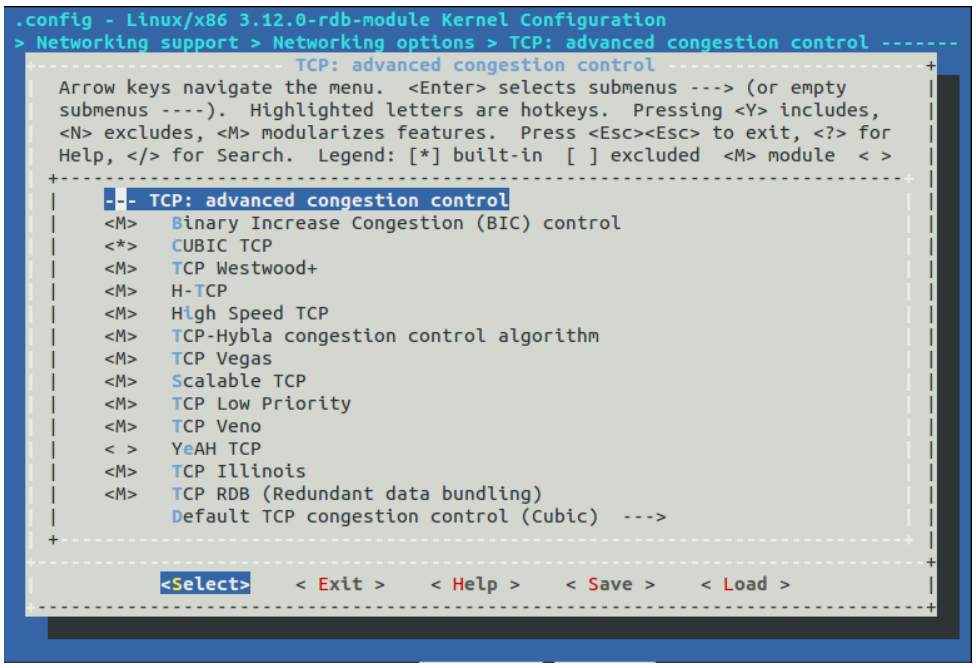


Figure 4.9: The Linux kernel config menu with a new RDB option

implement RDBv2 entirely as a kernel module, and only modify the kernel base files with the necessary changes for the entry points needed to execute the RDB code.

A major advantage of implementing RDB as a kernel module is that the code can be re-compiled much quicker without having to recompile the rest of the kernel. Another advantage is that the changes to the code can be loaded into a running kernel without rebooting.

The RDB implementation we present here, which we refer to as RDBv2, has been implemented as a kernel module configurable through the Linux kernel configuration, as shown by figure 4.9. In the figure, RDB is configured as a module, but it can also be compiled as part of the kernel, indicated by a star, like `CUBIC TCP`.

4.5.2 Congestion Control framework

The Linux CC framework is provided by the Linux kernel for implementing CCs as pluggable kernel modules. This makes it possible to easily separate the code for each CC implementation from the core of the TCP engine.

Code listing B.13 shows the entry points defined by the Linux CC framework, where `ssthresh` and `cong_avoid` are the only functions that the CC modules are required to implement (Code listing B.13, 11 and line 15).

4.5.3 RDBv2 implementation overview

The RDBv2-CC performs the following tasks:

- **Loss detection** based on processing incoming ACKs.
- **Congestion avoidance** performed through the `cong_avoid` hook.
- **ssthresh adjustment** after a CWND reduction.

The RDBv2 implementation is split across the following files:

- **`rdb.c`**
  Contains the initialization code for kernel module.
- **`rdb_cc.c`**
  Contains the CC code.
- **`rdb_skb.c`**
  Implements the functionality that tests if it is possible to bundle data in a SKB, and the functions to allocate and copy the redundant data into a new SKB.
- **`lib/rdb_tfrc.c`**
  This is the entry point for the TFRC-SP functionality utilized by the RDBv2-CC in `rdb_cc.c`.
- **`lib/loss_history.c`**
  This is where the functionality for keeping track of the loss event history, and the calculation of the loss event rate.
- **`lib/tfrc_equation.c`**
  This contains the function `tfrc_calc_x` that implements the throughput calculation from equation 2.2. Functionally, this is a direct copy of the file `net/dccp/ccids/lib/tfrc_equation.c`.

Figure 4.10 shows how the TCP engine calls the functions in RDBv2, through the Linux CC framework in code listing B.13.

The code that performs the data bundling, by modifying the SKBs, is called from `do_rdb` through the entry point `pkt_send`. The other three functions, `tcp_rdb_event`, `tcp_rdbcong_avoid_tfrc`, and `tcp_rdb_ssthresh`, are part of the RDBv2-CC.

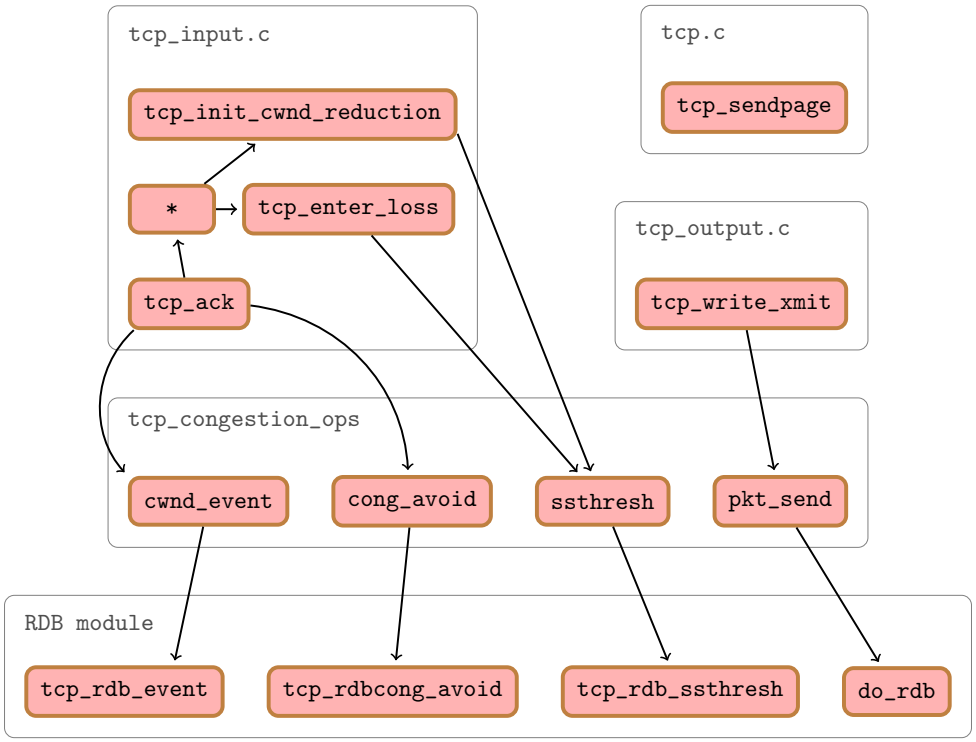


Figure 4.10: Call graph showing the relations between the TCP engine, the Linux CC framework, and the RDB module. The node containing a star indicates a call to a function that further calls the functions pointed to by the star node.

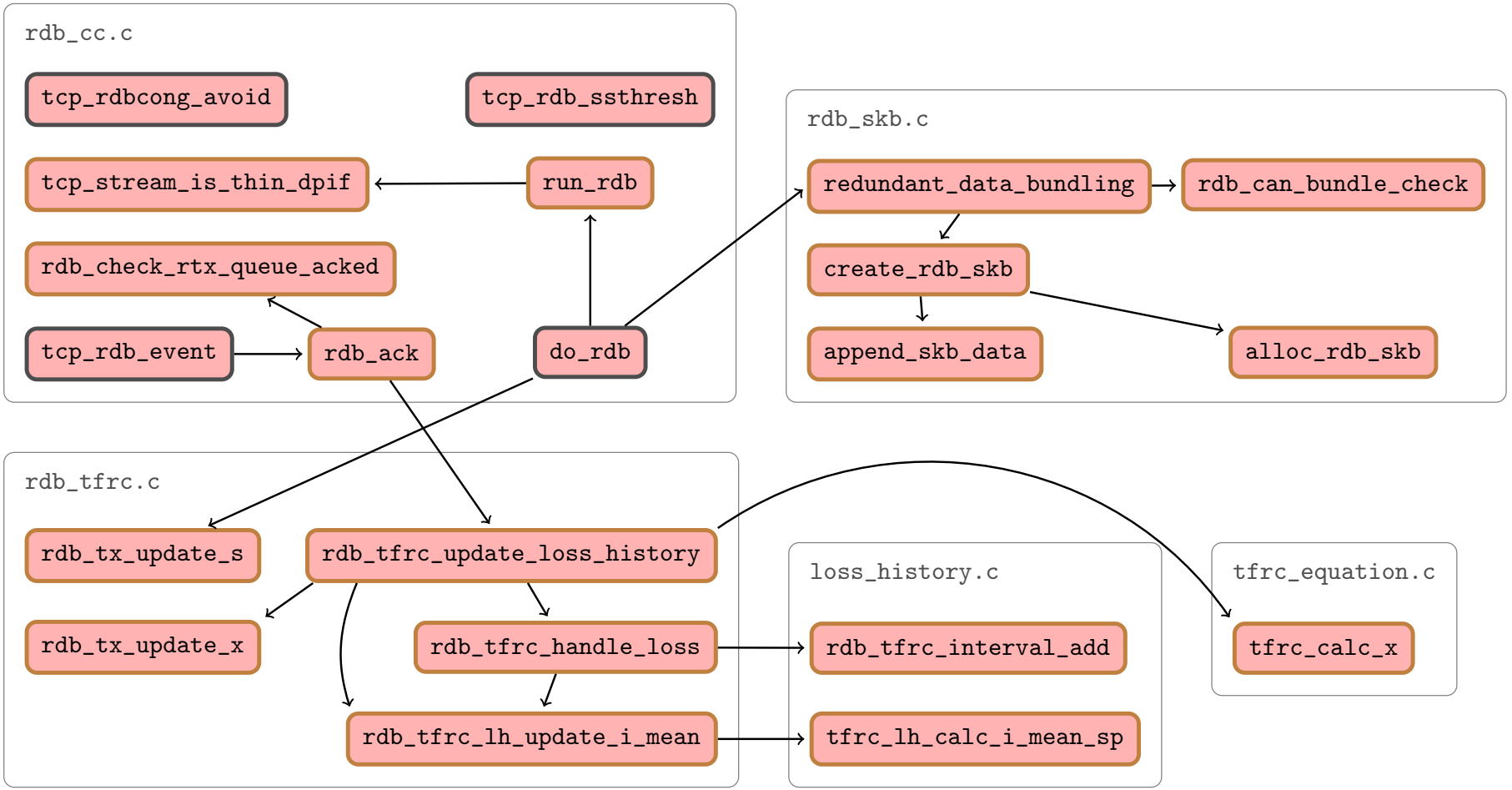


Figure 4.11: Callgraph for RDBv2

4.5.3.1 Processing ACKs

To calculate packet loss, the RDBv2 must analyse the state after each incoming ACK. While there is a specific hook for ack accounting `pkts_acked`, this hook is triggered during the cleanup process of the TCP output queue, after modifying the socket info. However, the hook `cwnd_event` (Code listing B.13 line 19), provides the necessary entry points we need. A requirement for RDB ACK analysis is that it can be performed before the default ACK handling mechanism (`tcp_ack`) cleans up the TCP output queue.

Code listing B.14 shows an excerpt from the function `tcp_ack`), where we see the two events, `CA_EVENT_FAST_ACK` (Line 11) and `CA_EVENT_SLOW_ACK` (Line 15), being triggered, which in turn call `cwnd_event`.

Detecting loss

The function `rdb_ack` (see code listing B.15) is called for each incoming ACK before the ACK has been processed by the TCP engine. It calls the function `rdb_check_rtx_queue_acked` (Code listing B.16) which traverses the TCP output queue to find how many packet were ACKed.

If any packets are detected as lost, the counter `rdb_ack_loss_count` is incremented with the lost packet count. If TFRC is enabled, it further calls the function `rdb_tfrc_update_loss_history`, which updates the loss interval with the new information.

Of the loss detection techniques described in section 4.2, only the simplest one has been implemented in RDBv2, where loss is detected by ACKs that cover multiple packets. This detection technique is sufficient for our testbed experiments, because we can be certain that no packet reordering occurs, and no ACK packets can be lost on the reverse path.

To limit the scope of the thesis, we therefore leave investigating this further as future work.

Performing TFRC calculations

After processing each incomming ACK, `rdb_ack` calls the function `rdb_tfrc_update_loss_history` (Code listing B.15 line 19), which updates the loss history (described in section 4.4.1.1).

If loss is detected, the function `rdb_tfrc_handle_loss` is called to update the loss history, before the average loss interval is recalculated with `tfrc_lh_calc_i_mean_sp` (Code listing B.17 line 13). If no loss is detected, the current (open) loss interval is updated with `rdb_tfrc_lh_update_i_mean`.

Further, `rdb_tfrc_update_loss_history` will call the function `tfrc_calc_x` (Code listing B.17 line 27) which is the implementation of the throughput calculation from equation 2.2. Finally, on line 36 in code listing B.17, `rdb_tfrc_update_loss_history` calls the function `rdb_tx_update_x` (shown in code listing B.18), which performs the task of calculating the new CWND value based on the send rate, according to equation 4.2 (as described in section 4.4.1.3).

4.5.3.2 Congestion avoidance

The CC hook `cong_avoid` is called by the TCP engine to grow the CWND. RDBv2 has two implementations, where `tcp_rdbcong_avoid_tfrc` is used when the TFRC loss response mechanism is enabled, and `tcp_rdbcong_avoid` is used otherwhise.

Both the implementations will perform the same basic tasks:

- **Update minimum RTT** (of the entire duration of the connection) if latest RTTM is smaller. The minimum RTT saved in `cong->minrtt_us` and used by `tcp_stream_is_thin_dpif`.
- **Restrict the CWND** if loss has been registered. When the ACK processing performed by `rdb_ack` detects a loss, a detects a loss, the CWND will be updated. If RDBv2-CC is used with TFRC enabled, the CWND is updated with the value calculated from the TFRC-SP throughput. If TFRC is not enabled, it will instead call the function `tcp_enter_cwr`. This function is used by the TCP engine to enter CWND reduction state.
- **Increase the CWND** if no loss has occurred. The CWND is increased using `tcp_reno_cong_avoid`, the `cong_avoid` handler for TCP New Reno.

Due to the negative effects caused by the changes to the function `tcp_is_cwnd_limited` in Linux kernel version 3.16 and forward, we decided to use the implementation from Linux kernel version 3.15 in the functions called by the `cong_avoid` CC hook (see code listing B.2).

4.5.3.3 Modify slow-start threshold

The Linux CC framework defines the entry point `ssthresh` (Code listing B.13 line 11), which is used by the TCP engine to set the slow-start threshold. The ssthresh hook implementation for TCP New Reno, `tcp_reno_ssthresh` (code listing B.22), sets the `ssthresh` to half the current CWND. To make RDBv2 less favorable for greedy streams, the rate at which the CWND grows could be reduced further compared to standard TCP.

An alternative implementation of the ssthresh entry point is shown in code listing B.23, where the ssthresh is set to $1/4$ of the CWND, instead of $1/2$. This is the same as used by TCP Nice, which is a CC designed to be less aggressive (*nicer*) than standard TCP, and is meant for tasks such as background transfers (*Mcdonald* and *Nelson* [2006, a3]).

4.6 Summary

We have described background for why we needed to develop the new RDB prototype implementation RDBv2, and how we chose to do it. RDBv1 was a modification to the core parts of the TCP engine, that implemented redundant bundling and sysctl variables for restricting how much bundling to perform.

By taking a different route, and developing RDBv2 as a CC module in the Linux CC framework, the RDB mechanism can be controlled more precisely by the custom CC algorithm we have implemented, referred to as RDBv2-CC. The main improvements RDBv2 brings over RDBv1 are:

- **DPIFL** is a dynamically adjusted PIFs limit, that is calculated based on the connection's RTT and the maximum allowed ITT specified for the connection. This enables a more precise control of which streams are classified as thin streams, and hence, which should be treated differently. This is achieved by basing the classification on the sender application's transmission patterns, instead of using a SPIFL that will differantiate between connections with different RTTs.
- **Loss detection** based on processing incoming ACKs. This solves the problem caused by RDB where loss events are hidden from the TCP engine.
- **The congestion control mechanism RDBv2-CC**, which is based on TFRC-SP. Based on the loss calculations, it calculates a throughput rate that oscillates less compared to standard TCP mechanisms that utilise the AIMD paradigme. This helps thin streams to maintain a higher CWND than regular TCP, while still trying to be TCP friendly.

In chapter 5, we evaluate the RDBv2 implementation that we have presented in this chapter.

Chapter 5

Evaluation of RDB prototype v2

In this chapter we will present how we have set up the experiments, and how we have evaluated the results. We will first described the metrics we have used for measuring the effects of the mechanisms we have developed. We will further describe the testbed and the test environment we have used, before we give some details on the tools we have developed and used to produce and analyse the results.

Finally, we will present the experiments we have performed, and describe some key results from the experiment tests.

5.1 Metrics for evaluating RDB

We define three metrics to measure the effect of RDB. ACK latency, TCP-friendliness and resource penalty. The ACK latency quantifies the gain in latency from the viewpoint of the sender. The TCP-friendliness quantifies the penalty incurred on competing traffic by the RDB mechanism, and resource penalty quantifies the resource overhead of the modified protocol.

The sections that follow will go into detail about the metrics.

5.1.1 Latency

As the main goal of RDB is to improve the latency for thin streams, the main metric we use for the evaluations is latency. The end goal is to improve the application layer latency, which is the time between a segment is sent from the application layer on the sender hosts, till the application layer on the receiver host can read the data segment. However, as we currently do not have a good way of measuring the application layer latency directly, we measure ACK latency instead.

5.1.1.1 ACK Latency

The ACK latency is the measurement of the time when the sender host sends the data for the first time till the sender host receives an ACK for the data. This is measured by analyzing the `tcpdump` captures of the network data on the sender side. When packets are lost, the in-order guarantee of TCP causes any data with

a higher sequence number to be delayed until the lost data has been retransmitted. By measuring when the data is ACKed, we include this extra delay in the latency numbers. The ACK latency calculations are performed with `Analysetcp` (described in section 5.3.4).

5.1.1.2 ACK Latency vs Application layer latency

The ACK latency resembles the application layer latency, but they differ in a few aspects. The application layer latency is the measurement from the time when the data is passed from the user space application to the kernel on the sender side, to the moment the data is passed from the kernel to the user space application on the receiver side.

The ACK latency differs, by measuring the time between when the data segment is first sent onto the network by the sender host, till the sender receives an ACK for that segment. While this may sound like very different measures, they effectively provide the same measurement data, with one key difference, namely the send buffering delay.

The ACK latency is calculated based on the data captures provided by tcpdump, which is fetched from the packets in the link layer on the sender host. Therefore, any processing delays in the higher layers of the sender host network stack are not included in these numbers.

When data is received in order, nothing but processing delay should delay the deliverance of the data to the application layer. Therefore, the application layer latency and ACK latency measure the same metric on the receiver side. However, on the sender side they can differ significantly. This is because the data can be delayed in the TCP output queue before it is passed to the lower layers. As long as Nagle's Algorithm is not disabled, it will cause delays for `send` calls with small data chunks. Another reason for delays is that the CWND prevents more data from being sent. This queuing delay is not reflected in the ACK latency data, but is an important part of the application layer latency measure.

For streams that send packets very seldom, this time difference *may* be negligible, as long as they are not limited by the CWND (and Nagle's Algorithm is disabled), but as soon as the sender starts to buffer data in the TCP output queue, this delay can be significant.

We use the ACK latency because that is the most accurate metric that measures the per-data-chunk *in order* delivery latency that is possible to calculate with the data provided from the `tcpdump` output traces.

To measure the application layer latency, we could use hooks in the `send` and `read` system calls to get the correct time stamps for when the data is sent and delivered. This would however require further work, and is out of the scope for this thesis.

5.1.1.3 Evaluating gains in latency

As we do not have tools for measuring the application layer latency, we need to compare the ACK latency values in the context of how the ITT values changes. If an experiment shows a decrease in the ACK latency values, it is also necessary to

take into account changes to the send buffering delay, which we indirectly can see from the ITT numbers. As we know that the user space application for each stream in two similar tests will send the same amount of data to the kernel, we know that if one test has lower average ITT, and consequently larger average payload per packet, it means the send buffering delay was higher.

5.1.2 TCP-friendliness

The TCP-friendliness is how well the RDB modifications play with current TCP functionality. There are many aspects to consider in this regard.

As described in section 2.5.3, the traditional definition of a TCP friendly network stream is one that does not outperform a TCP stream in terms of throughput. While this makes sense when the only goal is to obtain a fair bandwidth share, we must also consider the effect RDB has on the latency of competing TCP thin streams.

However, it is important to evaluate if the RDB mechanism behaves in a TCP friendly manner. The experiments presented in section 5.4.5, will therefore mainly consider the traditional throughput-based measure of TCP-friendliness.

5.1.3 Resources

By bundling extra data with the packets, RDB causes extra overhead on the network nodes that processes the packets. In scenarios where thin streams are limited by the CWND, the data gets buffered in the sender's TCP output queue causing larger and fewer packets to be sent. By using RDBv2-CC, the CWND will not so easily become the limiting factor, enabling the sender host to transmit the data immediately, meaning more packets gets sent over the network. This also causes extra overhead for the network nodes.

5.1.3.1 Calculating the cost of sending network data

To calculate the cost of sending a network packet we separate the processing tasks for an application *a* into:

- **Per-packet processing cost** α_a which is the processing cost for each packet independent of its size.
- **Per-byte processing cost** β_a which increases linearly for each byte in the packet.

From *Ramaswamy*, *Weng*, and *Wolf* [2004, a38] we have the following equation:

$$c_{a,l} = \alpha_a + \beta_a * l \tag{5.1}$$

Calculating the number of CPU instructions required to send a network message from user space is not easy, as it depends on the type of message and the kernel implementation. Protocols such as TCP, that provide services like reliable transmission through data checksums and in-order guarantee using sequence numbers, will require more processing than simpler protocols like UDP.

Measurements have shown that $12,000$ instructions is a fair estimate (*Gray* [1988, a39]). According to *Gray* [2000, a40] a rule of thumb for calculating the cost of sending a message is $10,000$ instructions per packet plus 10 instructions per byte, giving the following calculation (by equation 5.1) for a packet of 100 bytes:

$$\begin{aligned} \alpha_a = 10,000ins,\ \beta_a = 10ins,\ l = 100 \\ 11000ins = 10000 + 10 * 100 \end{aligned} \tag{5.2}$$

In the case of TCP, the `send` call performed by the application leads to a context switch into kernel mode, where the data is copied to the TCP output queue, either by utilizing the free space of an existing SKB, or by creating a new SKB. Protocol specific operations are also performed, such as calculating the sequence numbers and updating the socket struct.

A large part of the per-message cost is not strictly the cost of *sending* a network packet, but the entire sequence of events triggered by the call to `send` from user space. Therefore we can expect that sending 1000 bytes by issuing 1000 calls to `send`, with 1 byte for each call, will cost much more than doing 1 call with 1000 bytes. If Nagle's Algorithm is enabled, we can expect only 1 packet to be sent in both cases, but 1000 context switches, plus the instructions necessary to add each byte to the TCP output queue, will certainly be more costly in respect to CPU instructions.

With the introduction of different offloading schemes in the *Network interface cards* **(NICs)**, like segmentation offloading and checksum offloading, the number of required CPU instructions are reduced significantly. In the Linux kernel, a NIC driver can signal that it supports e.g. *TCP Segmentation Offload* **(TSO)** by setting the `NETIF_F_TSO` bit in the `netdev_features_t struct` in the socket `struct`. This requires that it supports TCP checksum offloading as well as *Scatter-Gather* **(SG)**, which is the ability of the NIC to produce TCP packets from a list of segmented memory locations. Instead of the kernel segmenting the data from user space into MSS sized segments placed in separate SKBs, this is left for the NIC to do. The TCP output engine it will instead add the data to the memory pages in newest SKB in the TCP output queue as long as the data in this SKB has not been sent. The total amount of data in these fragmented memory areas pointed to by one SKB may far exceed the MSS.

The SKBs have two variables, `nr_frags` and `frags`, where `nr_frags` is the number of fragments referred to in the `frags` array. Each element in this array, is a `skb_frag_struct` containing a reference to a memory page fragment `struct` and its length. When the NIC receives the SKB from the kernel, it will read the data from the memory pages, split the data into MSS sized segments, and create

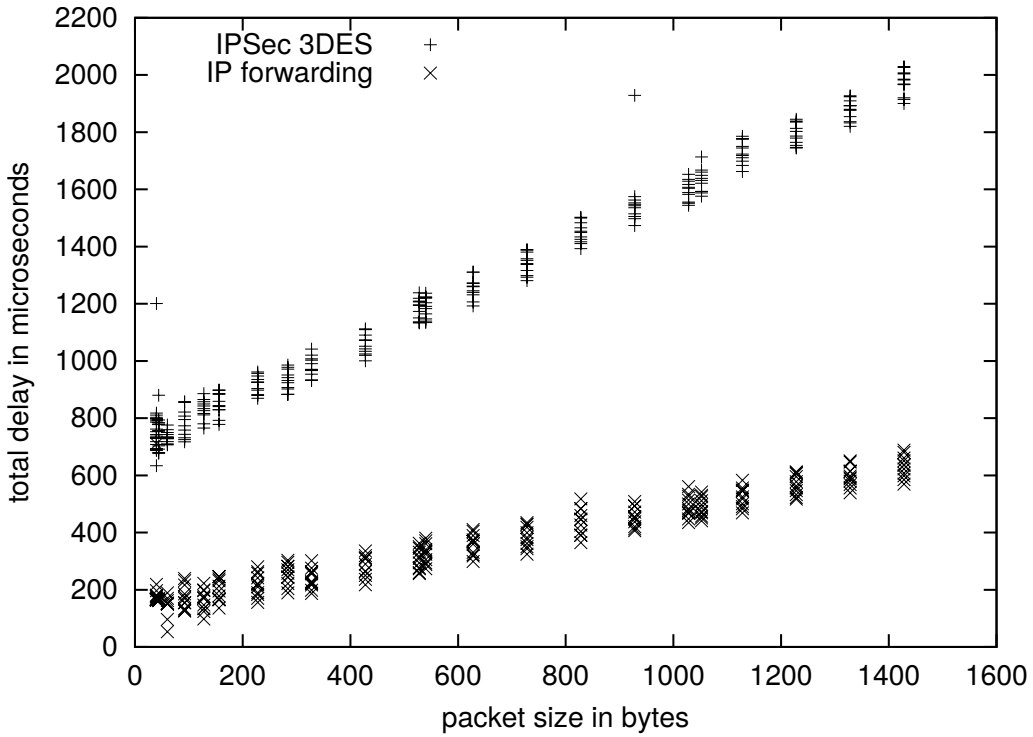


Figure 5.1: Packet processing time in a network router (*Ramaswamy*, *Weng*, and *Wolf* [2004, a38])

TCP packets. The reason the NIC must also support TCP checksum offloading in hardware is that if the TCP checksum is to be computed by the CPU, it is most efficient to combine the checksum operation with copying the data into the packet.

However, segmentation offloading has been shown to be useful only for transfers of large amounts of data *Foong* et al. [2003, a41].

Figure 5.1 shows the measurements of processing time in a router for IP packet forwarding and *Internet Protocol Security* **(IPSec)** packets passing through an IPSec tunnel. The forwarded IP packets require no processing of the payload, while the data in the IPSec packets need cryptographic processing. From the plot it is clear that the weights for α_a and β_a differ greatly between the two applications, where the per-byte processing cost (β_a) is more dominating for the IPSec packets than for the IP packets. One might expect the β_a for the IP packets to be zero, but even though the packet payload is not processed, it must be copied from the input queue to the output queue, and the cost of copying these bytes depends on the number of bytes.

5.1.3.2 Overhead of RDB

While RDB requires extra procedures to copy the redundant data into the packet to be sent, the CPU workload of transferring the data from user space into the TCP output queue is the same.

What may require extra CPU instructions is the RDBv2-CC that is required to traverse the TCP output queue regularly to perform loss detection, as well as the send rate calculation explained in section 4.4.1.3. The send rate calculation requires the continuous recalculation of the loss event rate, followed by the throughput calculation.

Even if RDB is a sender side modification, it may affect the receiver side's performance. With a regular TCP network stream, the receiver side will usually receive each packet in the correct order. Therefore, the receiver can expect that the sequence number of the next packet, to be equal to the sequence number of the last byte of the last data segment received, plus one. This will be the case unless packets are lost, or reordering occurs.

With RDB, this changes. As the first part of the data in RDB packets is old, the sequence number of the data will not be the expected sequence number, *unless* packets are lost, or reordering occurs.

The procedure that handles incoming packets in the TCP engine (`tcp_data_queue`), fails on the first three if-tests, before passing on the last if-test (see line 24). If the DSACK options is enabled, the receiver will also add the sequence range of the redundant data to the SACK option (see line 28). This causes extra packet processing and function calls compared to packets from a regular TCP stream.

Measuring the approximate overhead of using RDB could be done on a theoretical level, e.g. by using the amount of extra data RDB produces as input to equation 5.1. However, as the experiments show, it is often difficult to separate the effect of the bundling itself, and effects caused by how the behavior of the CC changes as a result of the RDB data. If RDB, together with the RDBv2-CC, cause more packets to be sent in total, the calculation must be adjusted to take the extra number of packets into account.

As we will see in some of the experiments, the RDB streams cause more loss for other thin streams in some scenarios. Some of the results show that the TCP thin streams end up sending fewer packets in total, with more payload per packet. This will actually cause less workload for the network nodes, as fewer packets have to transferred through the network.

As the bundling rate depends on many different criteria, such as the PIF limitation, the maximum limit of the sysctl variables described in section 4.3.2, the size of the data segments the application produces, and the interval between each `send` call. In addition comes the network characteristics, which will affect how easily the stream becomes network limited.

5.1.3.3 RDBv2 resource usage

To ensure minimum resource overhead for regular TCP streams that use a CC that does not implement the `pkt_send` entry point, the if-tests that test for `pkt_send` uses the keywords `unlikely` and `likely`. Normally, the compiler uses branch-prediction to optimize the machine code for the most likely branch path (if-else). These keywords are instructions that tell the compiler to optimize for a certain branch, causing the machine code to be optimized for branches specified using `likely`, and not for those specified with `unlikely`. By using `unlikely` when testing if `pkt_send` is defined, we ensure that the most probable branch (where it

is not defined), is the most efficient. However, when pkt_send is defined, the call to pkt_send will probably succeed, hence, we surround the call with likely.

We planned to profile the code to measure the resource usage of RDBv2, however, due to time constraints, we decided to leave this as future work.

5.2 Test environment

Simulation is a common way to perform experiments in network research. The ns-2 simulator created in the late 1990s, was for many years the de-facto standard tool used for within academic network research (*Riley* and *Henderson* [2010, a42]). Today, there are multiple alternatives, among them ns-3 set out to replace ns-2. ns-2, however, still has by far the richest TCP functionality (*Rosbach* [2012, a43]).

For the purpose of the experiments presented in this thesis, simulation is not a practical choice, as that would require implementing the modifications in the language of a simulator such as ns-2 in addition to the implementation in the Linux kernel.

To run tests that produce statistically significant results we need the tests to generate a certain amount of data. We may either have the tests last long enough to produce enough data, or make the tests produce data faster. By running multiple similar network streams, and aggregating the results, we are able to produce data faster.

5.2.1 Testbed setup

We have performed all the experiments in a lab testbed using Linux Debian 7. The testbed is set up with three sender hosts and one receiver host, with two bridges between them, as depicted in figure 5.2.

The two hosts used to send thin streams, rdbsender and wsender, are running Linux kernel version 3.16, and the host that sends greedy streams, zsender, is running Linux kernel version 3.12. The bridges, bridge1 and bridge2, are set up with rate control and network delay, respectively. Rate control is set up as depicted in code listing 5.1, and network delay is set up with netem as in code listing 5.2.

According to *Bufferbloat-project* [2014, b11], it is best to configure netem on a separate host, which is why we have used separate hosts for netem and rate control. We have also followed the guidelines of disabling any offloading mechanisms on the NICs.

Testbed configuration code

```
tc qdisc del dev eth1 root
tc qdisc add dev eth1 root handle 1: htb default 10
tc class add dev eth1 parent 1: classid 1:10 htb rate 5000kbit
tc qdisc add dev eth1 parent 1:10 pfifo limit %(BDP)s
```

Commands used to set up rate control on bridge1. %(BDP)s is replaced with the calculated BDP.

Code Listing 5.1: Setup of rate control with htb qdisc

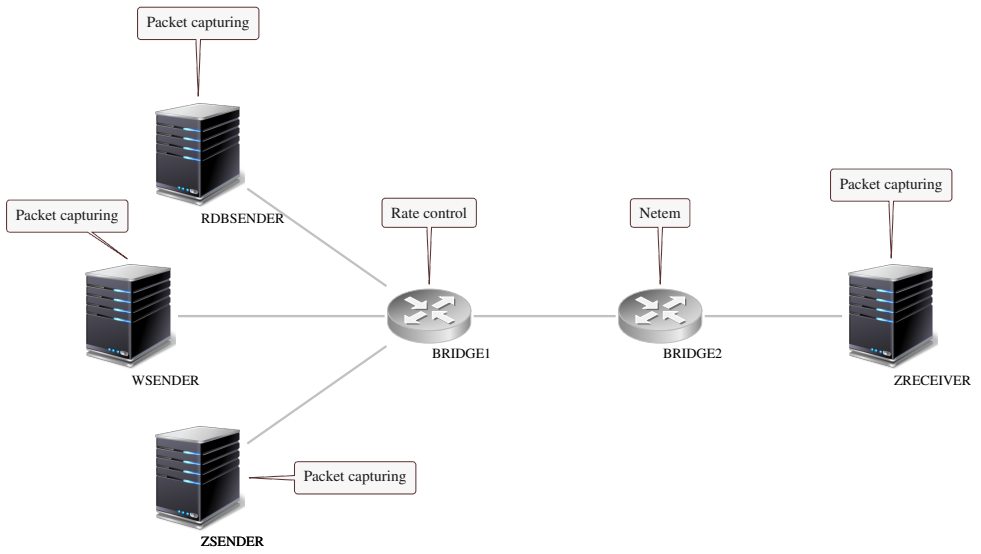


Figure 5.2: Testbed network setup

Testbed configuration code

```
tc qdisc add dev eth1 handle 1:0 root netem %(delay)s
tc qdisc add dev eth2 handle 1:1 root netem %(delay)s %(loss)s
```

Commands used to set up network delay `bridge2`. %(delay)s and %(loss)s is replaced with the values for the specific test being run.

Code Listing 5.2: Setup of network delay with `netem`

5.2.2 Challenges and pitfalls in testbed experiments

Choosing network parameters such as rate limit on the bottleneck and the RTT for the test machines is a much more complex topic than one might expect.

There are a multitude of variables that can effect the test results in unexpected ways, such as the configuration of the end hosts, the hardware of the NICs used (or the driver version), the router/switch hardware, and configuration of the bottleneck node.

There are many pitfalls when configuring the network nodes, such as the effect the clock interrupt rate of a hosts kernel may have on the behavior of buffer queues, and how configuring network emulation together with rate control may give unreliable results (*Bufferbloat-project* [2014, b11]).

Bufferbloat-project [2014, b11] advises to remove as much (potential) buffer bloat as possible, such as caused by offloading mechanisms in the kernel. We performed tests with different offloading mechanisms enabled and found there were

differences, but we had a hard time figuring out a system behind the different results. We ended up following the advice of disabling the offloading mechanisms on all hosts in the tested. These are *a) Generic Segmentation Offload* **(GSO)** *b) TCP Segmentation Offload* **(TSO)** *c) Generic Receive Offload* **(GRO)**, and *d) Large Receive Offload* **(LRO)**

5.2.3 Rate control

We have experimented with different parameters before finding a setup that works for all the different test setups we have used. The final setup is the result of a wide range of tests where we, at the end, believed we could trust the results to be correct.

5.2.3.1 Bufferbloat

The rate limit set for the bottleneck node will control the amount of data being sent through the node within a certain time. This however, brings up the question of buffer queue lengths and how that affects the network streams. The problem with traditional router queues is that different kinds of network streams share the same buffer. While queues are important to handle packet bursts, they also cause increased delays for delay-sensitive traffic. If the queue is too short, the node is unable to properly deal with fluctuations in the arrival rate caused by bursty traffic, but if they are too big they cause the buffers to stay consistently full, a problem known as bufferbloat (*Nichols* and *Jacobson* [2012, a44]). Even though the problem was identified decades ago, this issue is still relevant, and getting even more relevant due to today's cheaper memory leading to more available memory for queues in the routers, the increase in applications that need low latency, and the increase in bandwidth-hungry applications such as video streaming.

While there is no "correct" queue length, as it all depends on the network properties, and what you want to achieve with the queue, a standard approach is to base the queue length on the BDP (*Vu-Brugier* et al. [2007, a45]), which is what we have based our tests on.

5.2.3.2 Finding a rate limit setup

By basing the queue length on the bottleneck on the BDP as calculated by equation 5.3, we found it necessary to set the rate limit to a value that supports the number of simultaneous streams we want to test.

In the latency tests we restricted the tests to run up to $20 * 2 + 5$ simultaneous streams, and in the fairness tests, up to $20 * 3$ streams.

When first testing with a rate limit of 1000kbit/s, where the BDP suggests a queue size of of 13 packets, we experienced unreliable results in some of our tests. When comparing the results of a test with 5 greedy streams and 10 thin streams versus a test with 5 greedy streams and 20 thin streams, the difference in loss rates for the thin streams could be significant. We had expected the results to be fairly similar as we presumed the greedy streams would simply get a smaller share of the total bandwidth, however, that was not the case. With multiple greedy streams constantly trying to send at maximum rate, they continually have packets

in the queue on the bottleneck. In our test setup, there is a direct link between the sender hosts and the bottleneck (`bridge`), which means that the ability to absorb the packets depends solely on that one host. The thin streams are dependent on not arriving when the queue is full - the same queue the greedy streams are doing their best to fill up. With the queue being heavily used by greedy streams, the capacity for thin streams is not excessive, which explains why a doubling of the thin streams count would lead to unreliable results. The problem is not first and foremost the bandwidth limit being too small, but the small queue length being unable to handle that many network streams.

$$\begin{aligned} BDP &= \frac{kbit\_lim * 1000}{8} * (RTT * 10^{-3}) \\ QLEN &= \frac{BDP}{MTU} \end{aligned} \tag{5.3}$$

5.2.4 Network properties and stream characteristics

Testing thin streams with different characteristics on different types of network environments is important to find where RDB performs best, and finding the corner cases where it does not perform as well. The stream characteristics we have chosen are based on the typical streams observed by applications that produce thin streams (see table 2.1).

Different network environments will affect how RDB behaves. The two most important properties are the ITT in relation to the RTT, and the loss rates. For RDB, the ITT in relation to the RTT controls how much can be bundled. Therefore running tests where only the ITT is varied would in theory produce the results we are after. However, there is a problem with that theory, and that is the RTO timer, and how its minimum value is controlled by the static value `TCP_RTO_MIN`. With the default setting in the Linux kernel of 200, the RTO timer will never be less than 200 ms, independent of the RTT. On a connection with a low RTT, such as 10 ms, the RTO timer will therefore be very large relative to the RTT, compared to a connection with an RTT of 150 ms.

We have found it necessary to reduce the number of variables as much as possible to make the amount of test results manageable. We have therefore used an RTT of 150 ms for all the experiments, a value which gives room for a wide range of ITTs, and is is shown to be a good approximation of common RTTs in the Internet based on website RTT measurements (*Markussen* [2014, a34]).

To be able to run tests with a wide range of simultaneous streams, we decided on a rate limit of 5000kbit/s, which gives a pfifo queue length of 63 packets. However, due to rounding errors in our script, the value used for the tests is 60.

5.2.4.1 Variations on the ITT

When denoting the ITT used in the experiment, we use the notation $X : Y$, where X is the mean ITT, and Y is the variation. An ITT specified with 100 : 10, means

that the time to wait between every two send system calls, is given by randomly choosing a value from a normal distribution with mean 100, and standard deviation 10.

In section 5.3.3.1, we explain in detail the background for why we do this.

5.3 Tools

We have used a wide range of tools to perform the experiments and analyse the results. The tools we have developed in-house are explained briefly.

5.3.1 sshscheduler

`sshscheduler` is a python script we developed to run the experiments in the testbed. It it designed for easily generating a large number of jobs with different test properties. Implementing strict return value verification, and full logs for every command, makes it easy to identify errors and misconfigurations in the tests.

5.3.2 graph_r

`graph_r` is a collection of python scripts we have developed to analyse data and generate plots based on the results of the testbed experiments. By providing a directory containing pcap files from a test run, it will analyse the file names and extract the test parameters. Depending on which types of results is requested, it will run a set of commands to produce different types of results, like ACK latency, loss statistics, throughput, goodput, and CWND. Further, the results of plotted with R using the python wrapper rpy2.

5.3.3 streamzero

We have used `Streamzero` to generate network streams, a program developed specifically for the task of testing thin streams. It can set up multiple network streams that send data simultaneously, using either TCP or UDP. It is similar to `iperf`, but has a few extra features implemented for our use cases.

Important features that we have used when testing:

- **Send streams with defined packet size/ITT or bandwidth**
  When sending thin streams it does send calls to the kernel with small chunks of data to try to make the kernel send small packets, but it cannot guarantee small packets being sent, as the kernel may merge the data into bigger chunks depending on many factors and settings.
- **Specify randomly varying ITT and packet size with standard deviation.**
  This is important to avoid synchronous transmits when testing with multiple streams per host. By specifying an ITT using the notation $X : Y$, the time between each data segment is sent to the kernel is randomly chosen from a normal distribution of mean X and standard deviation Y. The random variation

is computed using a Box–Muller transform of the pseudo-random normally distributed numbers produced by `drand48`, as shown in code listing 5.3.

- **Specifying socket options**
  - Enable `tcp_thin_linear_timeouts` and `tcp_thin_dupack`
  - Disable Nagle's Algorithm
  - Enable RDB

Source Code

```
1 int get_negexp_val(int mean, int stdev) {
2     double rand = sqrt(-2 * log(drand48())) * cos(2 * M_PI *
          drand48());
3     return rand * stdev + mean;
4 }
```

Code Listing 5.3: Function in `streamzero` that generates pseudo-random numbers from a given mean and standard deviation.

Command example

```
root@rdbsender:~\$ ./streamzero_client -s 10.0.0.22 -p 6000 -v2 -A4 -W -j 3 -P
    22000 -d 5m -i 10:1 -S 300 -c 3  -G rdb -r -B1500

============================
   Registered intervals:
============================

Interval 1:
Duration:                    300 seconds
Bandwidth:                   240.00 Kbps ( 30.00 KBytes/s)
Packet size:                 300
Packets per second:          100.000000
Intertransmission time in ms: 10, stdev: 1

Active streams:  1, completed 0, goodput:   0.00  bps (  0.00  Bytes/s), time
    left: 00:04:59:999Number of streams started: 3
Waiting for streams to finish.
Connection 22002->10.0.0.22:6000 starting up
Connection 22003->10.0.0.22:6000 starting up
Connection 22004->10.0.0.22:6000 starting up
Active streams:  3, completed 0, goodput: 277.20 Kbps ( 34.65 KBytes/s), time
    left: 00:04:47:987
Caught signal SIGINT
Waiting for threads to finish...
Waiting for 3 threads.
Done waiting for threads!

3 threads out of 3 targeted arrived: 3 completed successfully, 0 errors and 0
    aborted.
Number of streams that timed out: 0
Done waiting for threads!

================================================================
                    STREAM SESSION ENDED
================================================================
    Running time 00:00:15:102
    Connections established:      3
    Total bytes transfered:       483300 (483.30 KB)
    Total packets sent:           1611 (Per stream: 537)
    Average session goodput:      256.00 Kbps ( 32.00 KBytes/s)
```

```
    Streams not finished cleanly: 0

root@rdbsender:~\$
```

Command example 5.1: Running `streamzero`

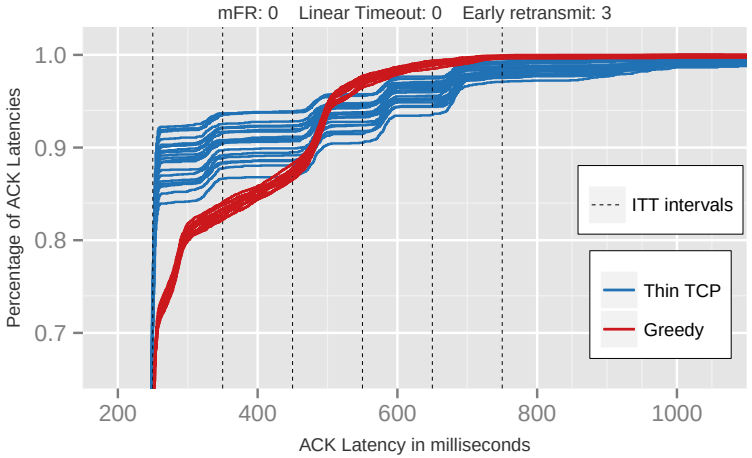
5.3.3.1 Why variate the ITT?

We noticed some strange results in our tests, regarding the loss rates of the individual streams from the hosts sending thin streams. For some tests the loss rates were varying to a great degree, where we expected them to be fairly equal. We tested with up to 40 simultaneous thin streams, and the more streams, the greater the variation of loss. After investigating the issue with many different tests, we found that the issue was caused by the thin streams sending data at regular intervals. When the total number of streams exceed a threshold, that depends on the bottleneck queue size, a certain number of streams will more often than others encounter a full queue, and hence lose the packet. When the ITT is not adjusted dynamically, the same streams will end up continually sending at a bad spot right after other thin streams have sent their packets.

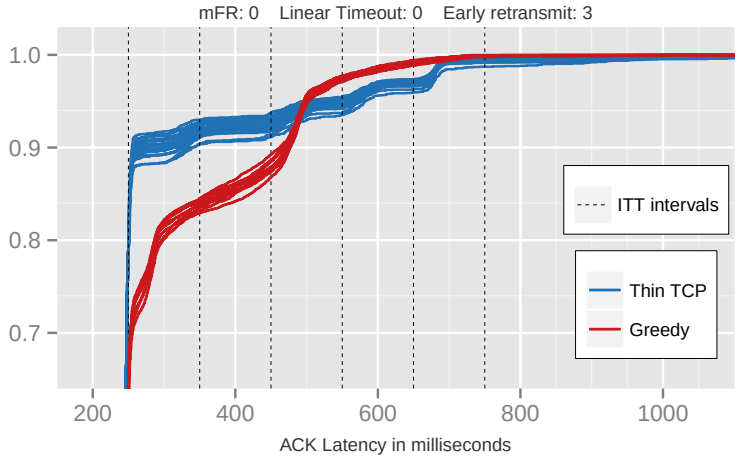
To illustrate this, the results in subfigure 3.2.(e) (from section 3.1) have been plotted together with the results from rerunning the test using an ITT of 100 ms, instead of 100:10 ms as in the original tests.

In these plots (figure 5.3) the latency is plotted for each stream, and not aggregated. Comparing the loss rates in the two the tables we see that the loss values of subfigure 5.3.(c) has a greater variation than subfigure 5.3.(d).

To avoid any synchronization issues with the thin streams we therefore decided to start the streams with a variable delay as well as using a random variation for the ITT of each thin stream.



(a) ITT: 100



(b) ITT: 100:15

Port	Est. Loss	Packets sent	Packets recv	Packet Loss	Byte loss
22000	4.11 %	2895	2777	4.08 %	4.12 %
22001	2.71 %	2915	2837	2.68 %	2.56 %
22002	3.00 %	2934	2847	2.97 %	2.85 %
22003	4.10 %	2878	2762	4.03 %	3.90 %
22004	3.99 %	2880	2767	3.92 %	3.75 %
22005	4.62 %	2877	2745	4.59 %	4.36 %
22006	2.67 %	2918	2841	2.64 %	2.50 %
22007	3.34 %	2871	2776	3.31 %	3.10 %
22008	3.26 %	2910	2816	3.23 %	3.04 %
22009	3.12 %	2917	2827	3.09 %	3.07 %
22010	3.68 %	2905	2799	3.65 %	3.63 %
22011	2.59 %	2938	2863	2.55 %	2.56 %
22012	2.77 %	2959	2878	2.74 %	2.75 %
22013	2.06 %	2965	2905	2.02 %	1.96 %
22014	2.59 %	2937	2862	2.55 %	2.47 %
22015	2.72 %	2941	2864	2.62 %	2.50 %
22016	3.45 %	2896	2798	3.38 %	3.35 %
22017	2.09 %	2965	2905	2.02 %	1.96 %
22018	2.28 %	2940	2873	2.28 %	2.21 %
22019	3.26 %	2914	2821	3.19 %	3.16 %
22020	3.66 %	2897	2791	3.66 %	3.53 %
Average		2916		3.0 %	3.1 %

(c) ITT: 100

Port	Est. Loss	Packets sent	Packets recv	Packet Loss	Byte loss
22000	3.00 %	2998	2909	2.97 %	3.00 %
22001	2.42 %	3018	2946	2.39 %	2.36 %
22002	2.59 %	3012	2935	2.56 %	2.61 %
22003	2.62 %	3010	2932	2.59 %	2.62 %
22004	2.63 %	3004	2926	2.60 %	2.52 %
22005	2.29 %	3008	2940	2.26 %	2.21 %
22006	2.72 %	3013	2933	2.66 %	3.01 %
22007	2.30 %	2998	2930	2.27 %	2.27 %
22008	2.28 %	3021	2953	2.25 %	2.24 %
22009	2.89 %	3008	2922	2.86 %	2.83 %
22010	2.36 %	3013	2943	2.32 %	2.27 %
22011	2.79 %	3014	2931	2.75 %	2.70 %
22012	2.71 %	3022	2941	2.68 %	2.63 %
22013	2.55 %	3020	2944	2.52 %	2.46 %
22014	2.79 %	3007	2924	2.76 %	2.68 %
22015	2.41 %	3024	2952	2.38 %	2.33 %
22016	2.30 %	3004	2936	2.26 %	2.31 %
22017	3.24 %	2998	2902	3.20 %	3.14 %
22018	2.45 %	3026	2952	2.45 %	2.43 %
22019	2.52 %	3015	2939	2.52 %	2.46 %
22020	2.75 %	3022	2939	2.75 %	2.68 %
Average		3012		2.6 %	2.6 %

(d) ITT: 100:15

Figure 5.3: Results showing the difference and variation in ACK latency and loss between static (100) and dynamic (100:15) ITT

5.3.3.2 Contributions

As part of this thesis, `Streamzero` has been in large parts rewritten and extended with a set of features:

- **Stream intervals** which is a way to specify stream characteristics that last for a certain period. With $-Id:10,i:300,b:2-Id:2,i:50,S:300:50$, two intervals are specified where the first interval has a 10 seconds duration, ITT or 300 ms, and bandwidth of 2kbps, which means it will adjust the size of the payload argument to `send` to match this throughput. The second interval lasts for 2 seconds, has an ITT of 50 ms and an average packet size of 300 bytes with a standard deviation of 50. The specified intervals will be run in a round-robin fashion until the global duration expires.
- **Specifying a Congestion Control Algorithm** to use for the streams. This was a strict requirement after implementing RDBv2. The alternative is to set RDB as the default CC, but that would lead to all the TCP connections using RDB which easily causes trouble when there are bugs in the code.
- **Support for UDP traffic** instead of TCP.
- **Bug fixes and performance improvements** allowing one sender host to run thousands of simultaneous threads.
- **Verify the data integrity with SHA-1**
  Implementing SHA-1 data verification between the server and client helped to identify problems during the development of RDBv2.

5.3.4 analyseTCP

`Analysetcp` is a tool developed by the authors of mFR, LT and RDBv1 for analyzing `tcpdump` trace (`pcap`) files. The most important feature of `Analysetcp` that we have used that other tools do not provide is analyzing both sender and receiver trace to calculate exact loss. `Analysetcp` can also calculate the variation of one-way latency for the packets by adjusting for clock drift on the two traces. It is also the only tool that can handle RDB packets, and differentiate between retransmitted and RDB data.
Important features that we have used when analyzing the results:

- **Calculating ACK latency based on ACKs**
  The ACK latency, described in 5.1.1.1, is the main metric we have used to measure the latency differences for different TCP variations.

- **Loss estimation based on retransmissions**
  This is simply the ratio of number of retransmission compared to number of packets sent. This gives a rough approximation of lost packets which are fairly correct in some scenarios. In others, such as when using RDB, the approximation is not useful as RDB reduces the number of retransmission drastically.

- **Calculating exact packet and byte loss**
  This has been one of the most important features we have used for this thesis. By comparing the data from the sender side pcap file to the data in the receiver file, the exact amount of lost packets and bytes is calculated. This has been very useful to get a precise overview of the network environment the network streams are tested in.

5.3.4.1 Contributions

As part of the work in this thesis, large parts of `Analysetcp` has been rewritten. A range of bugs have been implemented and fixed in the process of improving and stabilizing the code. At a certain point most traces were handled properly, and most new issues we encountered were caused by corner cases in the traces.

After comparing the results from `Analysetcp` with `tcptrace` and `wireshark/tshark`, we are confident that it produces reliable results.

- **Re-implementation of how the information for the data segments are stored**
  has enabled a more accurate analysis. Every data segment is structured in a `struct ByteRange` that contains all the relevant information. By storing the relative sequence numbers (first packet starts with sequence number 0) we could more easily compare the output with wireshark when debugging.
  We list here some of the variables from `struct ByteRange` and what feature they provide:
  - **packet_sent_count, packet_retrans_count and packet_received_count** giving a complete picture of the number of packets that were used to transfer this data.
  - **data_sent_count, data_retrans_count, data_received_count, rdb_count** providing the context for each transmit as well as how many times the data arrived on the receiver side.
  - **pcap timestamps for all the packets that sent this data segment** which makes it possible to use the correct timestamp when calculating the ACK latency. One might think that the correct timestamp is always from the last packet that sent the data, but the sender side may retransmit data even when it is not necessary, in which case using the timestamp from the last packet yields incorrect latencies.
  - **tcp timestamps for all the sent packets and the packet that was first received.** This makes it possible to identify exactly which of the sent packets were received.
  - **acked_sent, ack_count, dupack_count** to count the number of pure-ACKs sent for this sequence number, the number of times this sequence number was ACKed as well as the number of dupACKs.
- **Calculating RDB "hits" and "misses"**
  The `ByteRange` also has the variables `rdb_byte_hits` and `rdb_byte_miss` that are used to count the *hits* and *misses*. When RDB bundles data, *hits* is

the measure of how many of the RDB packets that filled a hole on the data sequence on the receiver side. *Misses* are the packets that contained bundled data which was already successfully delivered.

- **Ability to produce data from only parts of the trace**

- **Performance improvements**
  based on profiling with `valgrind`'s `cachegrind` tool as well as memory optimizations. The result is that `Analysetcp`, compared to earlier versions, performs much better both in terms of CPU time and memory usage, while producing more detailed data and statistics.

Command example

```
$ analyseTCP -s 10.0.0.12 -r 10.0.0.22 -p 5000 -f /root/bendiko/pcap/
    rdbsender_wsender_zsender_to_zreceiver/
    RDBSENDER_WSENDER_ZSENDER_RDB_vs_GREEDY_TFRC5/all_results/r5-rdb-itt10:1-
    ps120-ccrdb-da0-lt0-er3-dpif20-tfrc1_vs_w5-tcp-itt10:1-ps120-cccubic_vs_z5
    ..kbit5000_min5_rtt150_loss_pif0_qlen60_delayfixed_num0_rdbsender.pcap -g /
    root/bendiko/pcap/rdbsender_wsender_zsender_to_zreceiver/
    RDBSENDER_WSENDER_ZSENDER_RDB_vs_GREEDY_TFRC5/all_results/r5-rdb-itt10:1-
    ps120-ccrdb-da0-lt0-er3-dpif20-tfrc1_vs_w5-tcp-itt10:1-ps120-cccubic_vs_z5
    ..kbit5000_min5_rtt150_loss_pif0_qlen60_delayfixed_num0_zreceiver.pcap -A
Processing sent packets...
Using filter: 'tcp && src host 10.0.0.12 && dst host 10.0.0.22 && dst port
    5000'
Finished processing sent packets...
Processing receiver dump...
Using filter: 'tcp && src host 10.0.0.12 && dst host 10.0.0.22 && dst port
    5000'
Finished processing receiver dump...
10.0.0.12-22000-10.0.0.22-5000 : Failed to find timestamp for 1 out of 13869
    packets. These packets were at the end of the stream, so presumable they
    were just not caught by tcpdump.
10.0.0.12-22003-10.0.0.22-5000 : Failed to find timestamp for 1 out of 14090
    packets. These packets were at the end of the stream, so presumable they
    were just not caught by tcpdump.

Aggregated Statistics for 5 connections:
  Duration: 297 seconds (0.082500 hours)
  Total packets sent                              :      68152
  Total data packets sent                         :      68132
  Total pure acks (no payload)                    :         10
  SYN/FIN/RST packets sent                        :      7/5/0
  Number of retransmissions                       :        507
  Number of packets with bundled segments         :      20538
  Number of packets with redundant data           :      21043
  Number of received acks                         :      67143
  Total bytes sent (payload)                      :   36743144
  Number of unique bytes                          :   18530760
  Number of retransmitted bytes                   :     129984
  Redundant bytes (bytes already sent)            :   18212384 (49.57 %)
  Estimated loss rate based on retransmissions    :       0.74 %
------------------------------------------------------------
Receiver side loss stats:
  Number of packets received                      :      67319
  Packets lost                                    :        833
  Packet loss                                     :       1.22 %
  Bytes Lost (actual loss on receiver side)       :     546960
  Bytes Loss                                      :       1.49 %
  Ranges Lost (actual loss on receiver side)      :       2049
  Ranges Loss                                     :       1.53 %
------------------------------------------------------------
```

```
Payload size stats:
  Average of all packets in all connections     :        539
  Average of the average for each connection    :        550
  Minimum    payload (min, avg, max)            :         88,     100,     104
      bytes
  Average    payload (min, avg, max)            :        442,     551,     653
      bytes
  Maximum    payload (min, avg, max)            :       1448,    1448,    1448
      bytes
  Average for all packets in all conns          :          0 ms
-----------------------------------------------------------------
RDB stats:
  RDB packets                                   :      20538 (30.14% of data
      packets sent)
  RDB bytes bundled                             :    3453280 (9.40% of total
      bytes sent)
  RDB packet hits                               :        329 (1.60% of RDB
      packets sent)
  RDB packet misses                             :      20209 (98.40% of RDB
      packets sent)
  RDB byte hits                                 :      95232 (2.76% of RDB
      bytes, 0.26% of total bytes)
  RDB byte misses                               :    3358048 (97.24% of RDB
      bytes, 9.14% of total bytes)
-----------------------------------------------------------------
Latency stats (Average for all the connections)
  Minimum  latencies (min, avg, max)            :        150,     150,     150
      ms
  Average  latencies (min, avg, max)            :        702,     807,     907
      ms
  Maximum  latencies (min, avg, max)            :       1214,    2272,    4957
      ms
  Average for all packets in all conns          :        789 ms
-----------------------------------------------------------------
ITT stats (Average for all the connections)
  Minimum        ITT (min, avg, max)            :          0,       0,       0
      ms
  Average        ITT (min, avg, max)            :         17,      22,      29
      ms
  Maximum        ITT (min, avg, max)            :        843,    1341,    2703
      ms
  Average for all packets in all conns          :         21 ms
=================================================================


General info for entire dump:
  10.0.0.12: -> 10.0.0.22:5000
  Filename: /root/bendiko/pcap/rdbsender_wsender_zsender_to_zreceiver/
      RDBSENDER_WSENDER_ZSENDER_RDB_vs_GREEDY_TFRC5/all_results/r5-rdb-itt10:1-
      ps120-ccrdb-da0-lt0-er3-dpif20-tfrc1_vs_w5-tcp-itt10:1-ps120-
      cccubic_vs_z5..
      kbit5000_min5_rtt150_loss_pif0_qlen60_delayfixed_num0_rdbsender.pcap
  Sent Packet Count     : 68152
  Received Packet Count : 67319
  ACK Count             : 67143
  Sent Bytes Count      : 36743144
  Max payload size      : 1448
  Received Bytes        : 36196184
  Packets Lost          : 833
  Packet Loss           : 1.22227 %
  Ranges Count          : 67637
  Ranges Sent           : 133960
  Ranges Lost           : 2049
```

Command example 5.2: Running `analyseTCP`

5.3.5 tcpproberdb

`tcpproberdb` is a kernel module we have written which is based on the `tcpprobe` module available in the Linux kernel. This tool uses kprobes to initiate a callback for each received ACK to print different properties of the socket, like CWND and PIFs. kprobes enables code in the kernel to dynamically attach a callback routine to be called before entering a kernel function. This way of gathering the information should be more efficient than calling `getsockopt` with `TCP_INFO` argument from user space continually, and will also provide the information more consistently. The kprobe trap is guaranteed to be called for every call to the kernel function, while the `getsockopt` call will only return the information available at the time the user mode process is scheduled. Code listing 5.4 shows the script used before each test to load the module and capture the data. Example output from `tcpproberdb` can be seen in commands listing 5.3.

Source Code

```
1 #/bin/bash
2 PORT=${PORT:-0} # Default 0 == All
3 FULL=${FULL:-1} # On every ACK
4 FLUSH=${FLUSH:-1} # Flush on every packet
5 cd /root/tcp_probe_rdb
6 make reload V=1 PORT=$PORT FLUSH=$FLUSH FULL=$FULL
7 cat /proc/net/tcpproberdb > /root/tcpproberdb.out
```

The bash script used to fetch information of the TCP socket

Code Listing 5.4: Bash script used to run `tcpproberdb`

Command example

```
28.074420735 10.0.0.12:22018 10.0.0.22:5000 32 3444766184 3444765944 10
    2147483647 5792 302 29312 604 906 2 0, 0
28.174357401 10.0.0.12:22018 10.0.0.22:5000 32 3444766304 3444766064 10
    2147483647 5888 283 29312 604 887 2 0, 0
28.185576427 10.0.0.12:22019 10.0.0.22:5000 32 1150720319 1150720079 10
    2147483647 5792 302 29312 604 906 2 0, 0
28.275249416 10.0.0.12:22018 10.0.0.22:5000 32 3444766424 3444766184 10
    2147483647 5888 266 29312 604 870 2 0, 0
28.285576217 10.0.0.12:22019 10.0.0.22:5000 32 1150720439 1150720199 10
    2147483647 5888 283 29312 604 887 2 0, 0
28.287799684 10.0.0.12:22020 10.0.0.22:5000 32 922659059 922658819 10
    2147483647 5792 302 29312 604 906 2 0, 0
28.393789342 10.0.0.12:22018 10.0.0.22:5000 32 3444766544 3444766304 10
    2147483647 5888 252 29312 600 852 2 0, 0
28.408976188 10.0.0.12:22019 10.0.0.22:5000 32 1150720559 1150720319 10
    2147483647 5888 266 29312 604 870 2 0, 0
```

Command example 5.3: Example output from `tcpproberdb`

5.3.6 Modifications to netem

During the development of RDBv2, we modified `netem` by adding a new loss mode. By default, `netem` supports specifying random packet loss. While this useful for experiments, it was not as useful while developing RDBv2. To easily reproduce a reliably and simple loss pattern, we extended `netem` with a fixed loss mode. The patch for the Linux kernel and `iproute2` can be studied at appendix C.2.

5.4 Experiments

We have performed three experiments with different configurations, where we test the latency by calculating the ACK latency, and the TCP-fairness by comparing the throughput and goodput.

- **Experiment 1** is designed to test the effect RDB has on the packet latency, and especially how it alleviates the HOL blocking issue. To isolate the effect RDB has on the latency as much as possible, the tests in this experiment are run in a more controlled fashion compared to the other experiments. To avoid the unpredictability of a shared bottleneck, we use only one host to generate traffic, and induce different uniform loss rates using `netem`.
- **Experiment 2**, which we call the latency experiments, is designed to test the effect of RDB in a more realistic scenario. In addition to the thin streams that we want to measure, a set of Greedy streams are run simultaneously through the same network path, to cause congestion.
- **Experiment 3** uses the same configuration as experiment 2, but adds a limitation on the amount of data the RDB streams can bundle with each packet.
- **Experiment 4** is set up to test the TCP-friendliness of RDBv2. This is done by using a transmission pattern that tries to increase the throughput by bundling redundant data on streams that are not *thin*.

From the results in appendix A.2 we have picked out some key results that show interesting effects or patterns that we think is the most important when evaluating RDB.

5.4.1 Reading the plot results

We analyse the goodput, throughput and ACK latency and plot the aggregated values of all the network streams sent from each host along with some key parameters in tables below each plot.

The results are presented in three different plot types, goodput, throughput, and ACK latency. All the plots contain information about the test parameters at the top, which identifies which test they describe. Each plot type is accompanied by a table containing a range of relevant data. We will go through the three different plot types, and show an example of each type.

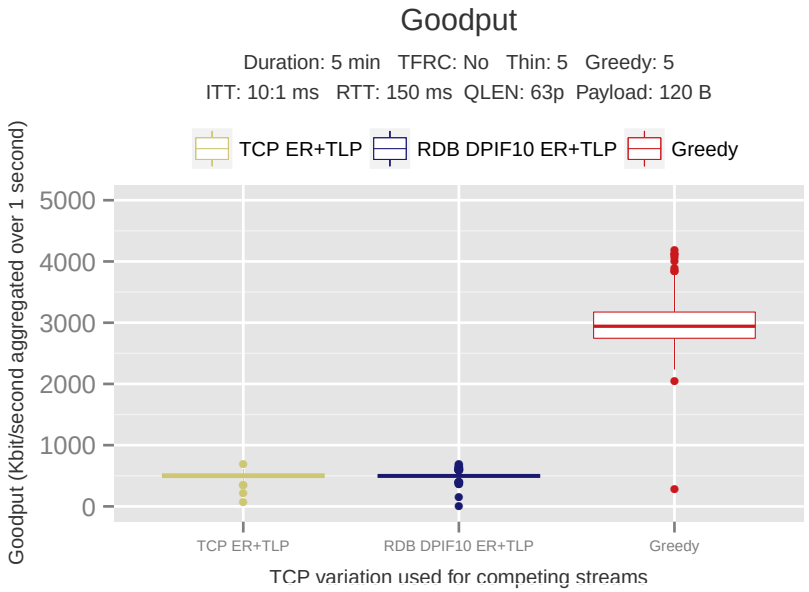
The plot examples that follow are from a test with three sender hosts, where the two hosts `wsender` and `rdbsender` start 5 thin streams each, which competes

against 5 greedy streams produced by `zsender`. The ITT for the thin streams is 10:1, the RTT between all the sender hosts and the receiver host is 150 ms. The rate limit is set at 5000kbit, with pfifo queue of length 63 packets. The thin streams write 120 bytes each ITT.

5.4.1.1 Goodput

The goodput is measured on the receiver side `tcpdump` trace, and is aggregated over all the streams of each type. The goodput is presented in a boxplot, where the goodput rate is aggregated for all the streams. The columns in the table below each goodput plot is as follows:

- **Name:** is the TCP/RDB variation used for these streams.
- **Host:** is the name of the host generating these streams.
- **Streams:** is the number of streams this host is sending.
- **ITT (avg):** is the minimum, average and maximum values of the average ITT calculated for each stream sent by this host. In figure 5.4, the ITT values of $24/28/32$ is the result of calculating the average ITT over all the packets of each stream separately. The stream with the lowest average ITT value had 24 ms, and the highest average was 32 ms. 28 ms is the value calculated by taking the average of the average for each stream.
- **Payload (avg):** is the minimum, average and maximum values of the average payload calculated for each stream on this host. These values are calculated like the ITT values.
- **CWND:** is the CWND values (Mean,Minimum,Q1,Median,Q3,Maximum) calculated from the values of all the streams sent from this host.



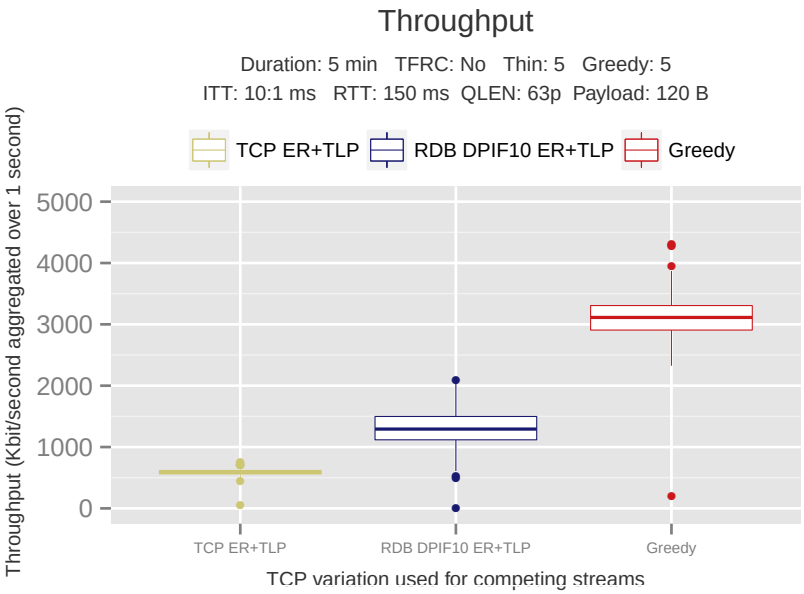
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	24 / 28 / 32	315/367/407	10/1/7/10/13/30
RDB DPIF10 ER+TLP	rdbsender	5	28 / 33 / 41	1001/1036/1057	10/1/6/9/12/28
Greedy	zsender	5	16 / 19 / 22	1446/1447/1447	15/1/10/13/18/51

Figure 5.4: Goodput plot example

5.4.1.2 Throughput

As with goodput, the throughput is measured on the receiver side `tcpdump` trace, and is aggregated over all the streams of each stream type. The throughput includes the IP and TCP header size in the calculation. The table columns are as follows:

- **Sent p:** is the number of packets sent for the streams.
- **Lost p:** is the number of packets lost for the streams.
- **Retr p:** is the number of packets retransmitted by the streams.
- **RDB p:** is the number of packets containing RDB data sent by the streams.
- **Payl B:** is the total number of payload bytes sent by the streams.
- **Unique B:** is the total amount of unique bytes, i.e., payload bytes subtracted all redundant bytes.
- **Rdn B:** redundant bytes is the payload that was retransmitted after the first transmission, i.e., the payload bytes subtracted the unique bytes. This included RDB data and data from retransmissions.
- **Retr B:** is the total amount of payload in retransmissions.
- **Loss % (B/p):** is the average byte loss rate, and the average packet loss rate of the streams on this host.
- **DupACKs:** is the total number of dupACKs received by the sender side of all the streams.

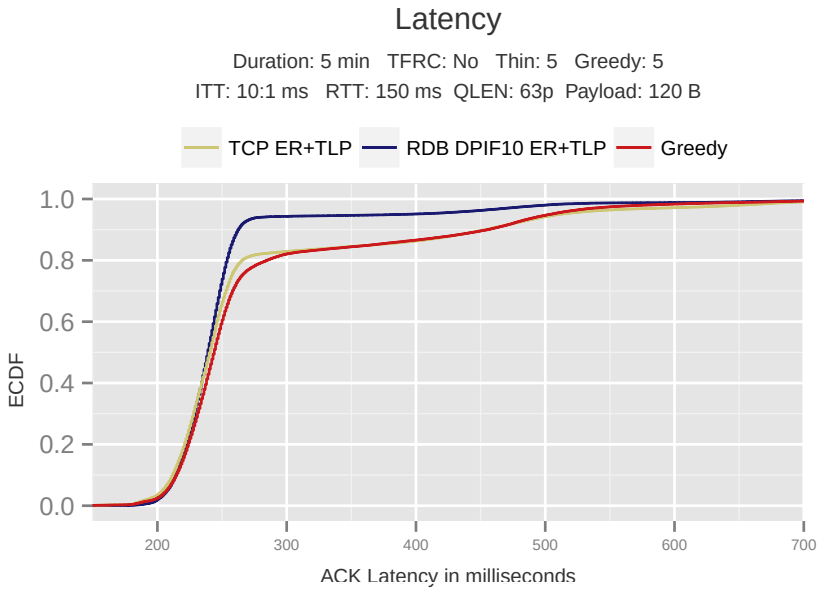


Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.6 K	1340	1340	0	19.2 M	18.8 M	334 K	334 K	1.74 / 2.55	8708
45.2 K	1262	428	31.7 K	46.8 M	18.6 M	28.2 M	307 K	2.76 / 2.79	2385
79.7 K	1764	1764	0	115 M	113 M	2.6 M	2.6 M	2.21 / 2.21	13174

Figure 5.5: Throughput plot example

5.4.1.3 Latency

This is the aggregated ACK latency for all the streams calculated from the sender side tcpdump trace. In the table in figure 5.6 we can see the average and maximum ACK latency, a long with a range of percentiles chosen to give an overview of the most important parts of the distribution. While the maximum ACK latency value can be interesting, it cannot be used to evaluate which group performs the best.



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
279	241	246	253	266	454	486	511	534	587	651	696	1228	1678	1782
253	239	244	248	254	263	270	391	441	470	500	638	1030	1481	1775
280	244	250	258	285	455	483	504	517	537	576	665	864	1113	1522

Figure 5.6: Latency plot example

5.4.2 Experiment 1 - Latency tests with uniform loss

The tests we have performed in this experiment are set up with a uniform loss, enforced by `netem` on the host `bridge2`. The test setup for this experiment differ significantly to the setup of the experiments in the following sections. The tests in this experiment have only one sender machine, that sends one type of stream. This is to rule out unpredictable effects and variations, such as for the loss rate, caused by competition with greedy streams at the bottleneck.

Therefore, each plot presented in this section shows the result of three different test runs, where each test run generated streams with one type of the TCP variations in the parameters list below.

In contrast to the experiments in the following sections, which use the DPIFL (presented in section 4.1), the bundling rate of the RDB streams in these tests restricted by a SPIFL. This is to isolate the effect the different PIF limits have on the bundle frequency. The SPIFL is enforced by the function `tcp_stream_is_thin_spif` (code listing B.8) described in section 4.3.2, the the customized version of `tcp_stream_is_thin` in the current kernel.

By using a uniform loss distribution, instead of enforcing congestion with greedy cross traffic, we are able to avoid natural variations to the latencies caused by queuing delay. This allows us to fulfill the main goal of this experiment, which is to illustrate the effect of HOL blocking described in section 3.1.1, and how RDB alleviates this issue.

5.4.2.1 Test parameters

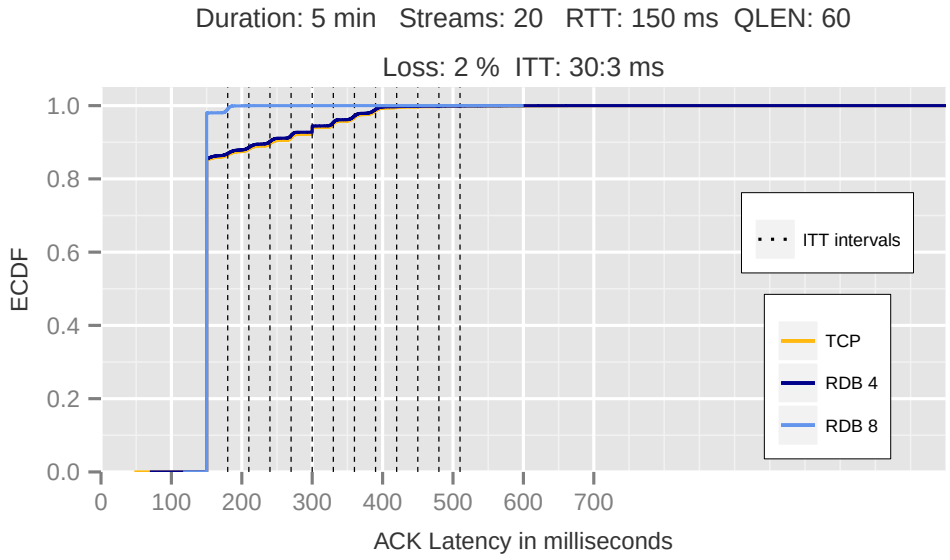
The test parameters for this experiment is as follows:

`Network`

- **RTT:** 150 ms
- **Uniform loss:** 0.5%, 2%, 5%, and 10%

`rdbsender`

- **Stream type:** Thin
- **Stream count:** 20
- **Payload:** 120 bytes
- **ITT:** 30:3, 50:5, and 100:10 ms
- **TCP variation:**
  - **TCP New Reno** Linux kernel version 3.16 with ER+TLP
  - **RDB** with SPIFL4
  - **RDB** with SPIFL8



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	172	150	150	150	150	150	243	330	361	393	572	871	1433
RDB 4	170	150	150	150	150	150	239	328	359	390	532	630	1158
RDB 8	151	150	150	150	150	150	150	150	150	180	186	231	567

Figure 5.7: Experiment 1: Loss 2%, ITT 30:3

In appendix A.1 the complete test setup can be seen in table A.1, along with the full set of plots for the experiment.

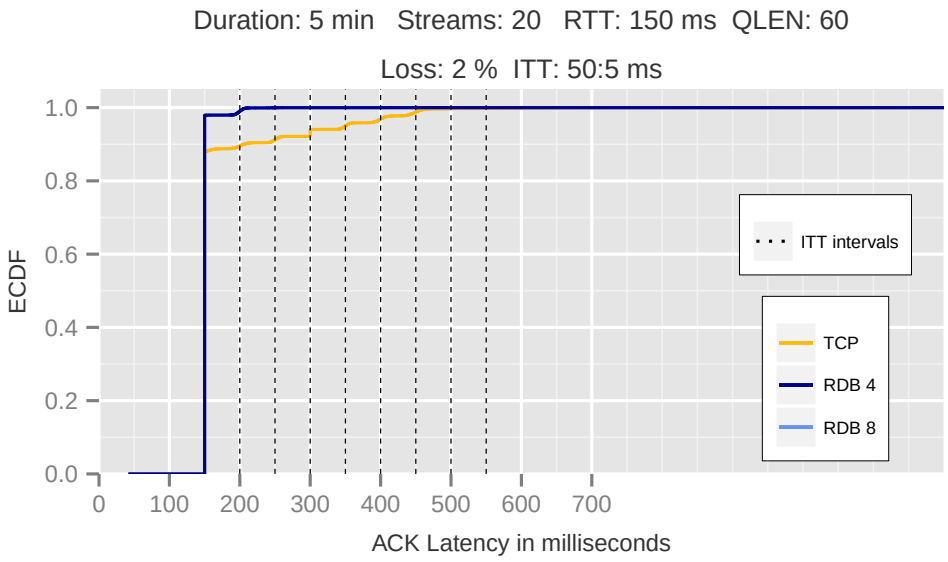
5.4.2.2 Key results

Figure 5.7 shows the results from running thin streams with ITT of 30:3 and 2% loss. We see how the ACK latency for the RDB streams with SPIFL of 8 (RDB 8), is exceptional compared to the other streams, with a 97^{th} percentile of 150 ms. The RDB streams with SPIFL of 4 (RDB 4) have the same latencies as the TCP streams, simply because the ITT of 30:3 results in a PIF value above the threshold of 4.

In figure 5.8, where the streams send with an ITT of 50:5, we can see that the ACK latencies for RDB 4 and RDB 8 are similar. In these tests, the higher ITT enables the RDB 4 streams to bundle redundant data on all the packets.

In these plots we can observed the same *stair pattern* as found in the preliminary experiments presented in section 3.1.

From figure 5.8 we can see that the 97^{th} percentile for both the RDB 4 and RDB 8 streams, is 150 ms. This means that the ACK latencies for 97% of the packets sent were not affected at all by HOL blocking. The small step for the RDB streams between $150 - 200$ on the x-axis, is the packets that were lost, but were recovered by the redundant data bundled with the next data packet. The ACK latency for these packet are therefore increased by 50 ms, the same as the ITT.

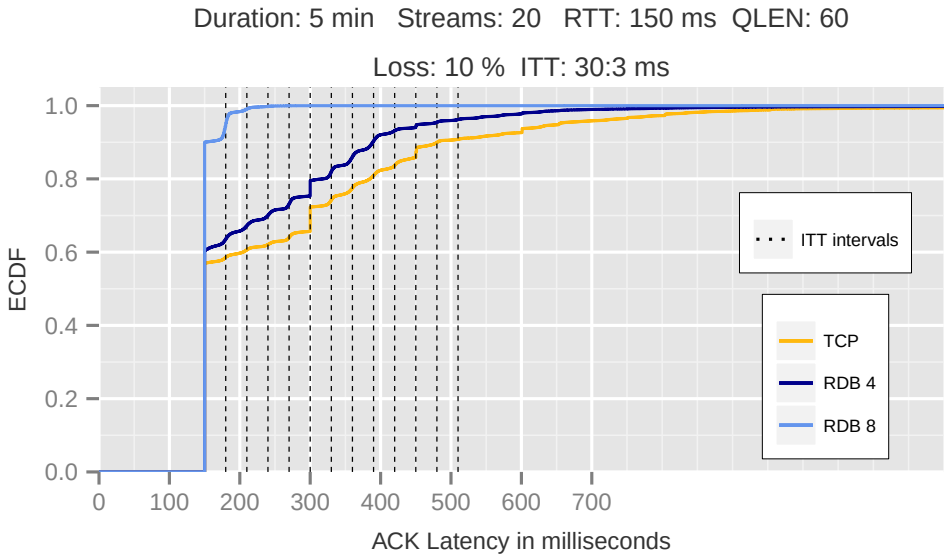


Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	172	150	150	150	150	150	206	350	401	452	610	1261	1505
RDB 4	152	150	150	150	150	150	150	150	150	200	213	1238	1503
RDB 8	152	150	150	150	150	150	150	150	150	200	210	1238	1504

Figure 5.8: Experiment 1: Loss 2% ITT 50:5

In figure 5.9 we see a mix, where the RDB 8 can bundle on all packets, and RDB 4 can bundle on some, but not all. In this plot, the stair pattern is even easier to spot. Looking at the ACK latency values for RDB 8, we can see that 90% of the packets were unaffected by the packet loss. The 97^{th} percentile of 83 ms, shows that most packets that got lost were recovered by the redundant data in the following packet.

The RDB 4 streams were only allowed to bundle on some of the packets, which give them better ACK latencies than the TCP streams, however, the difference to RDB 8 is striking.



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	270	150	150	205	300	385	480	654	775	956	2150	2852	4095
RDB 4	227	150	150	150	240	318	389	463	551	717	1511	5495	9650
RDB 8	154	150	150	150	150	150	150	180	183	209	250	813	3154

Figure 5.9: Experiment 1: Loss 10% ITT 30:3

5.4.2.3 Summary

The results of the tests in this experiment show us that RDB causes improvements on the ACK latency. When the SPIFL is high enough, allowing RDB to bundle for most packets, the ACK latency for most packets are unaffected by the HOL blocking phenomenon.

We see that HOL blocking is reduced greatly on the streams that are allowed to bundle on all the sent packets. All losses of one packet on the RDB streams cause only a HOL blocking of $ITT * 1$ as can be seen in (section 5.4.2.1), where the 95^{th} percentile for the RDB 8 streams are 150 ms ($RTT = 150$), and 99^{th} percentile is 180 ms ($RTT + ITT * 1$.

The very small second step we can see in figure 5.9, between 180 and 210 on the x-axis, is caused by the loss of two packets in a row, where the third RDB packet recovers the lost data, resulting in an increase of $ITT * 2$ for the ACK latency. This is confirmed by the 99^{th} of 209 ms.

All in all, we see that the latency numbers for the streams that are allowed to bundle are substantially lower, where the most important improvement is that the potential HOL blocking on the receiver side's application layer is reduced to a minimum.

Explaining the stair pattern

The stair pattern can be explained by a few different factors, mainly the lack of properties that normally cause a high degree of variation to the ACK latencies.

The most important factor is the these tests have virtually no queuing delay, causing most events to be triggered on very strict points in time. In addition,

disabling Nagle's Algorithm and TCP delayed acknowledgment gives the sender and receiver less room to send data packets and ACKs at their own pace. The result is that the ACK latencies for the RDB streams is $RTT + ITT * L$ where L is the number of times the packet is lost.

When a packet is lost for the TCP streams, the packets that follow are held back on the receiver side due to HOL blocking. These packets have exactly $ITT * i$ difference in ACK latency where i is their place in the queue. Their ACK latency is dependent on when the missing data is retransmitted, but this is also the result of a "deterministic" chain of events [1]. It is either caused by *a)* an RTO, which is strictly dependent on the RTT measurements, which does not vary much for these tests, or *b)* a fast retransmit that is triggered by the reception of ACKs, which are also coming at a "deterministic" time of one RTT after a packet is sent.

[1] Deterministic in the context of a best effort packet switched network.

5.4.3 Experiment 2 - Latency tests with greedy cross traffic

We have performed a set of tests to measure the latency of different thin streams in a network with competing greedy streams. This is to create a network environment that is more realistic than the uniform loss induced by `netem`.

These tests are set up with three senders, where the hosts `rdbsender`, and `wsender` send thin streams, and host `zsender` sends greedy traffic to cause congestion. The `zsender` is set up to run a constant number of greedy streams of 5 for all tests. This is to ensure a certain amount of traffic through the bottleneck, leading to a more realistic scenario for the thin streams.

5.4.3.1 Test parameters

The test parameters for this experiment is as follows:

`Network`

- **RTT:** 150 ms
- **Rate limit:** 5000kbit
- **Queue:** pfifo with 63 packet limit

`rdbsender`

- **Stream type:** Thin
- **Stream count:** 5, 10, and 20
- **Payload:** 120 bytes
- **ITT:** 10:1, 30:3, 75:7, and 100:10 ms
- **TCP variation:**
  - **TCP New Reno** from Linux kernel version 3.15 with ER+TLP
  - **RDB** with DPIFL10
  - **RDB** with DPIFL10 + TFRC
  - **RDB** with DPIFL20
  - **RDB** with DPIFL20 + TFRC

`wsender`

- **Stream type:** Thin
- **Stream count:** 5, 10, and 20
- **Payload:** 120 bytes
- **ITT:** 10:1, 30:3, 75:7, and 100:10 ms
- **TCP variation:** TCP New Reno from Linux kernel version 3.15 with ER+TLP

zsender

- **Stream type:** Greedy
- **Stream count:** 5
- **TCP variation:** system default TCP Cubic with ER+TLP.

In appendix A.2 the complete test setup can be seen in table A.2, followed by the full set of plots for the experiment.

5.4.3.2 Key results

The amount of congestion varies to a great deal in the different tests in this experiment. While the tests cover a range of different environments, some are more relevant than others. As the number of thin streams increase in the different tests, we see the effect more network congestion has on the streams.

The tests with 32 $(16 + 16)$ thin streams competing with 5 greedy stream on a 5Mbit bottleneck, can be considered an edge case, at least when the ITT is as low as 10:1, where we are at the limit of whether they can be considered as *thin*.

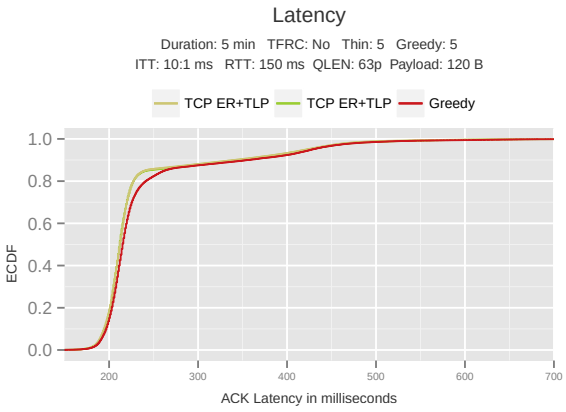
We start out with pointing out some key points which is relevant when analyzing any of the results in the experiments.

Reading the latency values

As explained in section 5.1.1.2, the ACK latency differs from the application layer latency, and does not provide the complete picture in many cases. Consider the plots in figure 5.10, which show the results from tests with $5 + 5$ thin streams, performing `send` system calls with an ITT of 10:1, against 5 greedy stream. We refer to these tests as `E2-5-10:1`, where the test with only TCP streams is named `E2-5-10:1-TCP`, and the test with RDB streams is named `E2-5-10:1-RDB`.

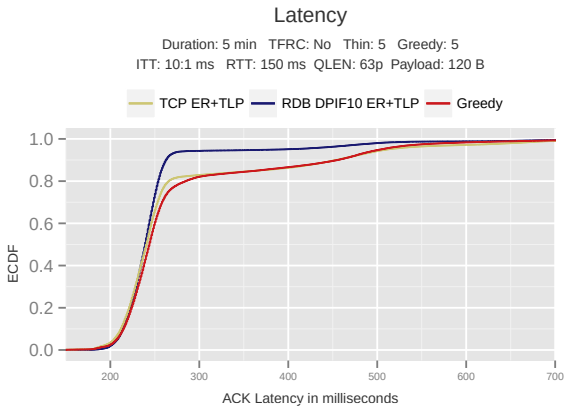
Looking at the latency plots for `E2-5-10:1-TCP` (subfigure 5.10.(a)) and `E2-5-10:1-RDB` (subfigure 5.10.(b)), we clearly see a difference in how the ACK latency for the RDB streams (blue line) sticks out. At a quick glance, it looks like the RDB mechanism provides a much better latency. There are, however, a few important things that can be easy to miss, which may put the results in a different light.

By looking at the ITT values in the table below the goodput plots, we see the difference in queuing delay on the sender side that is not reflected in the ACK latency numbers. For the `E2-5-10:1-TCP` goodput plot (subfigure 5.11.(a)), the ITTs average at 19 and 20 ms, while in the RDB plot, they average at 28 ms for the TCP streams, and 33 ms for the RDB streams. This matches well with the payload sizes in each of the tables, showing a big difference between the two tests. In `T-5-10:1-RDB`, the streams from both the thin-stream hosts send fewer packets (higher ITT) with larger payload, compared to `E2-5-10:1-TCP`. Naturally, the packets sent by the RDB streams are much larger due to bundling, however, the amount of new data in each packet is also bigger.



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
236	212	216	220	227	340	395	422	434	449	466	506	674	1052	1319
237	212	215	220	228	349	401	426	438	453	474	529	712	1025	1152
242	215	220	226	241	356	408	428	439	454	477	530	727	1088	1152

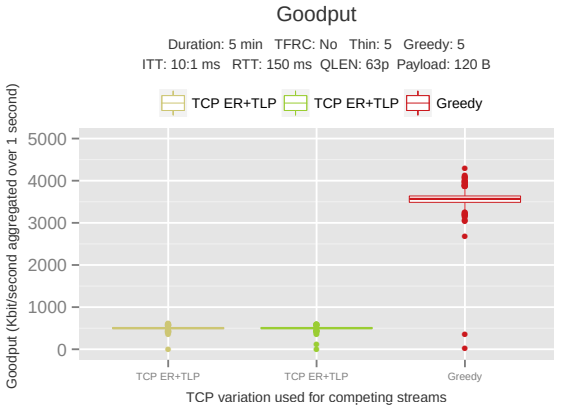
(a) E2-5-10:1-TCP



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
279	241	246	253	266	454	486	511	534	587	651	696	1228	1678	1782
253	239	244	248	254	263	270	391	441	470	500	638	1030	1481	1775
280	244	250	258	285	455	483	504	517	537	576	665	864	1113	1522

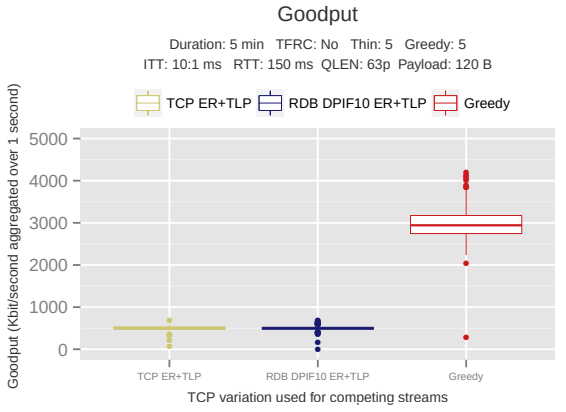
(b) RDB-5-10:1

Figure 5.10



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	17 / 19 / 21	223/246/275	13/1/9/13/17/28
TCP ER+TLP	rdbsender	5	18 / 20 / 22	229/256/285	13/1/9/12/16/28
Greedy	zsender	5	15 / 16 / 16	1446/1447/1447	14/3/10/13/16/36

(a) E2-5-10:1-TCP



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	24 / 28 / 32	315/367/407	10/1/7/10/13/30
RDB DPIF10 ER+TLP	rdbsender	5	28 / 33 / 41	1001/1036/1057	10/1/6/9/12/28
Greedy	zsender	5	16 / 19 / 22	1446/1447/1447	15/1/10/13/18/51

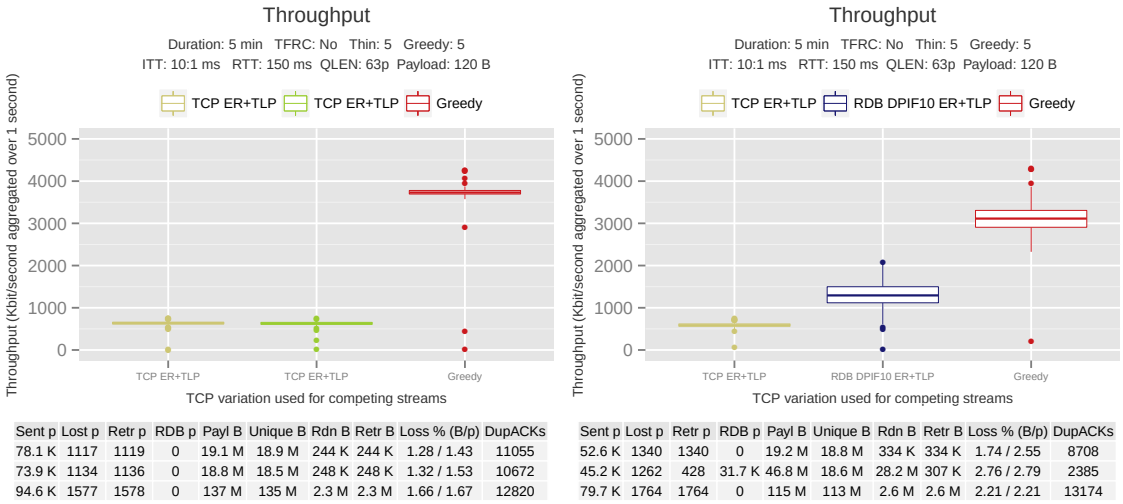
(b) E2-5-10:1-RDB

Figure 5.11

RDB reducing the latency for all streams due to higher loss

If we look closer at the latency values, and compare the results for the two tests, we can see an important difference. While the RDB streams in E2-5-10:1-RDB have better latencies than the TCP streams, the latencies for all the streams are also increased compared to E2-5-10:1-TCP. The average for any of the streams in E2-5-10:1-RDB is higher than for E2-5-10:1-TCP, and so are most of the percentiles. The exceptions are the 90^{th}, 93^{th} 95^{th} percentiles for the RDB streams, which are the lowest in both tests.

This is also reflected in the steepness of how the ACK latency distribution increases in the latency plots, where a steeper increase reflects less variation in the ACK latencies. More congestion on the bottleneck node causes greater variation in queuing delay, and hence, greater variations of the ACK latencies. The more



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
78.1 K	1117	1119	0	19.1 M	18.9 M	244 K	244 K	1.28 / 1.43	11055
73.9 K	1134	1136	0	18.8 M	18.5 M	248 K	248 K	1.32 / 1.53	10672
94.6 K	1577	1578	0	137 M	135 M	2.3 M	2.3 M	1.66 / 1.67	12820

(a) `E2-5-10:1-TCP`

Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.6 K	1340	1340	0	19.2 M	18.8 M	334 K	334 K	1.74 / 2.55	8708
45.2 K	1262	428	31.7 K	46.8 M	18.6 M	28.2 M	307 K	2.76 / 2.79	2385
79.7 K	1764	1764	0	115 M	113 M	2.6 M	2.6 M	2.21 / 2.21	13174

(b) `E2-5-10:1-RDB`

Figure 5.12

congested scenario in `E2-5-10:1-RDB` shows a less steep increase, meaning that more of the values fall within a greater range up to around 0.8. This stands in contrast to the latency plots presented in experiment 1 (section 5.4.2.2), where the queuing delay is non-existent, and the steepness is a straight vertical line.

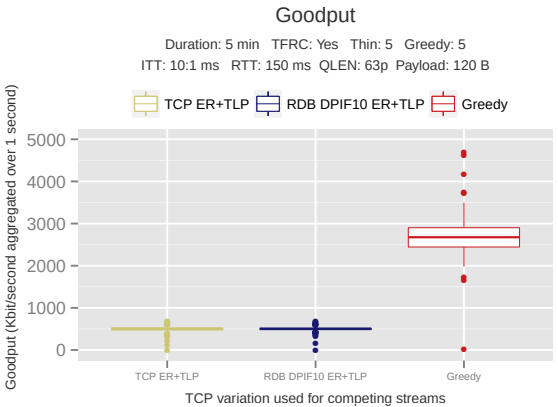
These values can be explained by comparing the loss rates in the throughput plots of `E2-5-10:1-TCP` (subfigure 5.12.(a)) and `E2-5-10:1-RDB` (subfigure 5.12.(b)). Due to the increased payload of the RDB streams caused by redundant bundling, the total throughput is greater, as seen in the throughput plot of `E2-5-10:1-TCP`. This leads to the rate limit discarding more packets, causing increased loss rates for all the streams. While the number of lost packets for the TCP streams only increase from 1117 in `E2-5-10:1-TCP` to 1340 in `E2-5-10:1-RDB` (20% increase), the relative packet loss rate increases from 1.43% to 2.55% (78% increase).

As the relative packet loss rate is the main factor controlling the CWND, this explains the lower send rate in terms of packets sent. This is confirmed by the values in the CWND column, where we see how the thin streams in `E2-5-10:1-TCP` have slightly higher values, with a median around 13, compared to a median of 10 in `E2-5-10:1-TCP`.

Effects of TFRC

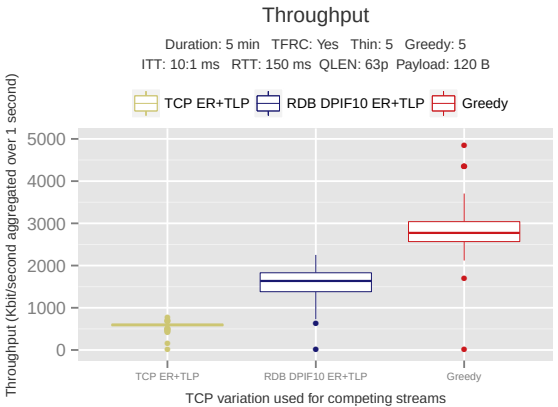
The plots of the test with TFRC enabled, referred to as `E2-5-10:1-RDB-TFRC`, show some interesting effects which confirms the reasoning in section 5.4.3.2. From the goodput plot of `E2-5-10:1-RDB-TFRC` (subfigure 5.13.(a)), we can see that the CWND values average at 12, which is the same as the greedy streams, and higher than for the RDB streams in `E2-5-10:1-RDB` (which is 10).

The increase in the CWND allows the RDB streams to maintain more PIFs, leading to sending a total of $59.5K$ packets, compared to 45.2 in `E2-5-10:1-RDB`. We also see this reflected in the ITT values, which average at 25 ms in `E2-5-10:1-`



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	26 / 28 / 31	333/365/398	10/1/7/10/13/28
RDB DPIF10 ER+TLP	rdbsender	5	24 / 25 / 26	888/959/1023	12/1/8/11/14/86
Greedy	zsender	5	19 / 21 / 23	1446/1447/1448	12/1/9/11/15/42

(a) E2-5-10:1-RDB-TFRC



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.8 K	1290	1292	0	19.2 M	18.8 M	385 K	385 K	2.00 / 2.44	8764
59.5 K	1504	449	42.9 K	56.9 M	18.6 M	38.4 M	190 K	2.51 / 2.53	3617
72.2 K	1614	1616	0	104 M	102 M	2.3 M	2.3 M	2.23 / 2.24	11405

(b) E2-5-10:1-RDB-TFRC

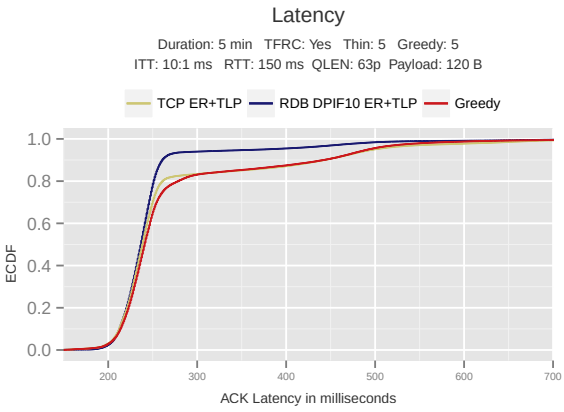
Figure 5.13

RDB-TFRC compared to 33 ms in E2-5-10:1-RDB. This leads to a lower relative packet loss rate for the RDB streams in E2-5-10:1-RDB-TFRC, even though they send more redundant data in total (increase from 28.2M in E2-5-10:1-RDB to 38.4M in E2-5-10:1-RDB-TFRC).

From the latency plot E2-5-10:1-RDB-TFRC (subfigure 5.13.(c)) we see that the average ACK latencies are slightly lower compared to the values for E2-5-10:1-RDB.

When comparing the results from the tests with and without TFRC, we see the following effects:

- the RDB streams in E2-5-10:1-RDB-TFRC get a higher average CWND, leading to more packet sent in total, and a lower average ITT, compared to E2-5-10:1-RDB. This gives less send buffering delay.
- The ACK latency measurements in E2-5-10:1-RDB and E2-5-10:1-RDB-TFRC are approximately the same, with a slight improvement for both thin streams in the test with TFRC enabled.
- The higher CWND in E2-5-10:1-RDB-TFRC in compared to E2-5-10:1-RDB does not affect the TCP streams negatively. However, from the results of E2-5-10:1-TCP we can see that the TCP streams perform worse when competing with the RDB streams. The average ITT increases, the number of packets sent decreases, and the ACK latencies are higher.



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
274	238	244	250	261	443	475	499	516	548	616	674	1018	1725	2420
250	237	241	246	251	261	272	374	422	453	484	557	934	1592	1871
278	241	247	255	279	442	473	492	504	522	552	623	931	10073	13590

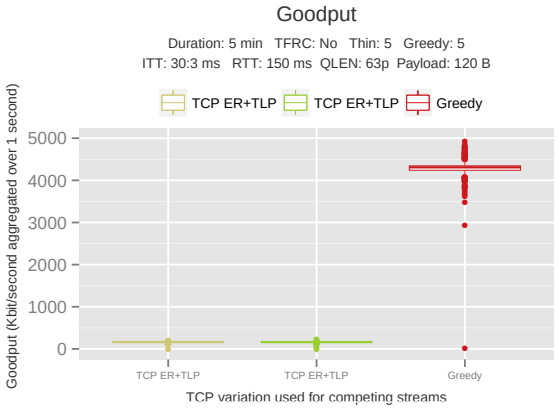
(c) `E2-5-10:1-RDB-TFRC`

Figure 5.13

Less congested scenario with ITT of 30:3

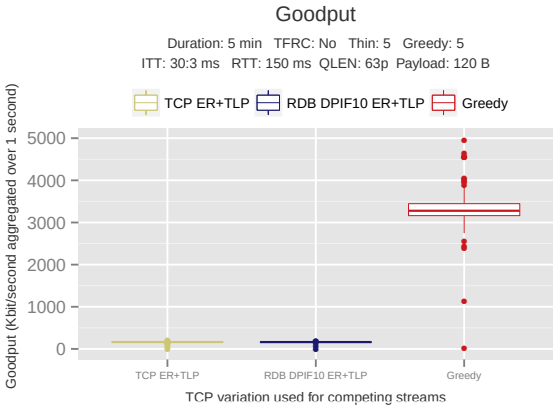
In the tests `E2-5-30:3`, where the ITT has been raised to 30:3, we see how the streams perform in a less congested scenario. Comparing the goodput plots in figure 5.14 of `E2-5-30:3-TCP` and `E2-5-30:3-RDB`, we see that the ITT and CWND values are almost identical, indicating that the negative effect the RDB streams have on the competing TCP streams, is not significant.

From the throughput plots (figure 5.15) we can see that both the thin stream groups send approximately the same number of packet in both tests, however, looking at the greedy streams, the number of packets sent decreased from $113K$ in `E2-5-30:3-TCP`, to $87.8K$ in `E2-5-30:3-RDB`. The most revealing information,



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 32	126/127/130	11/1/8/12/13/14
TCP ER+TLP	rdbsender	5	31 / 31 / 31	126/127/128	11/3/9/12/13/13
Greedy	zsender	5	12 / 13 / 14	1446/1447/1448	18/6/14/17/21/50

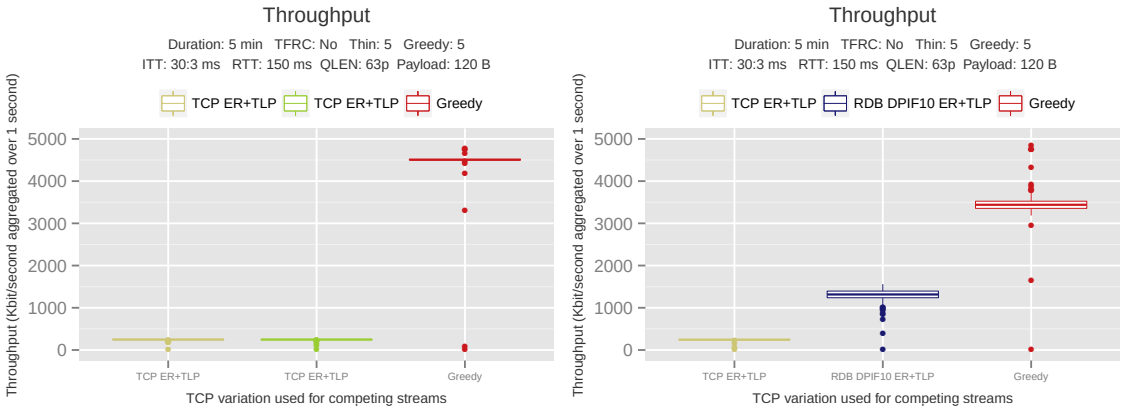
(a) `E2-5-30:3-TCP`



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 32 / 32	128/130/130	10/2/8/11/13/14
RDB DPIF10 ER+TLP	rdbsender	5	32 / 33 / 33	1015/1017/1018	10/2/8/11/13/14
Greedy	zsender	5	15 / 17 / 18	1447/1447/1447	14/4/11/14/16/37

(b) `E2-5-30:3-RDB`

Figure 5.14



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.4 K	540	542	0	6.2 M	6.1 M	65.5 K	65.5 K	1.06 / 1.12	4205
47.8 K	532	534	0	6.1 M	6.0 M	64.3 K	64.3 K	1.06 / 1.11	4062
113 K	1175	1177	0	164 M	162 M	1.7 M	1.7 M	1.04 / 1.04	13889

(a) `E2-5-30:3-TCP`

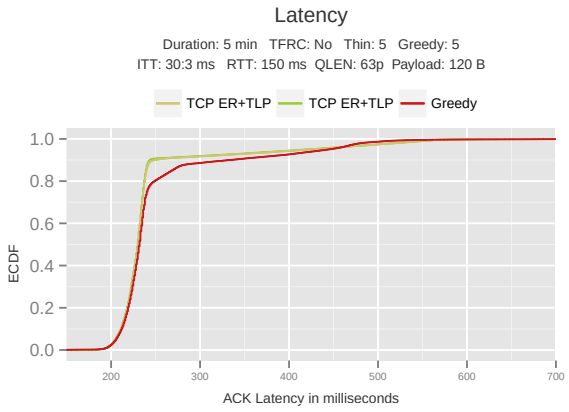
Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
47.5 K	611	613	0	6.2 M	6.1 M	74.7 K	74.7 K	1.21 / 1.28	4615
45.2 K	568	2	45.2 K	46.0 M	6.0 M	40.0 M	0	1.31 / 1.26	
87.8 K	1458	1460	0	127 M	125 M	2.1 M	2.1 M	1.66 / 1.66	12759

(b) `E2-5-30:3-RDB`

Figure 5.15

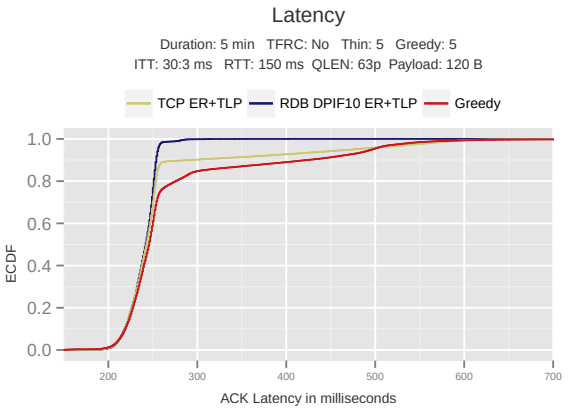
that shows why the RDB streams have better latencies than the competing TCP streams, can be found in the column "Retr p" in the throughput plots. The number of packet retransmissions is reduced from 534 in `E2-5-30:3-TCP`, to 2 in `E2-5-30:3-RDB`.

From the latency plots in figure 5.16 we see that the RDB streams have considerably better ACK latencies than the competing TCP streams, where the 99^{th} percentile of the RDB streams is 278 ms, compared to 551 ms for the TCP streams. When comparing the ACK latencies for the TCP streams in `E2-5-30:3-TCP`, with the TCP streams in `E2-5-30:3-RDB`, we see the unfortunate effect the RDB streams have on the competing TCP streams. The ACK latency values for the TCP streams are generally higher in `E2-5-30:3-TCP` compared `E2-5-30:3-RDB`. This may be explained by the increase in retransmissions, from 542 to 613, due to the higher loss rates caused by the RDB streams.



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	230	233	235	238	249	353	425	458	489	520	549	616	841	1152
245	230	233	235	237	244	347	419	454	487	519	551	664	890	1152
253	232	235	238	249	333	406	444	459	469	481	510	691	900	1152

(a) `E2-5-30:3-TCP`



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
262	243	247	250	253	293	410	475	504	534	561	589	733	1154	1777
240	242	246	249	251	254	255	256	257	258	259	278	308	385	1153
274	246	250	254	276	423	477	497	504	516	538	575	785	1015	1151

(b) `E2-5-30:3-RDB`

Figure 5.16

5.4.3.3 Summary

In the tests performed in this experiment, we see that the streams utilizing RDB consistently achieves better latencies than the competing thin stream using TCP. At the same time, we see that the competing streams are affected negatively, to a varying degree depending on the test scenario. In some cases, such as the test `E2-5-10:1`, both the competing TCP thin streams, and the greedy cross traffic experience a performance reduction when competing with RDB, where the throughput of the greedy streams is reduced significantly, and the latency of the TCP thin streams is heavily increased. In other cases, such as `E2-5-30:3`, the increased aggressiveness of the RDB streams mostly affect the greedy cross traffic, who's throughput is greatly decreased, while the competing TCP thin stream experience no increase in the average ITT, and less severe increases to the latencies compared to the `E2-5-10:1` tests.

An important key finding in the `E2-5-10:1-RDB` tests, is that while the RDB streams achieve better latencies than the competing TCP thin stream, the increased aggressiveness produces more loss and congestion. The result is that, compared to `E2-5-10:1-TCP`, the latencies for all the streams increase, including the RDB streams, as shown by an increased mean ACK latency.

In these tests we do not see any trace of the stair pattern we found in in the tests with uniform loss. This is caused by the continuous variations in the queuing delays on the bottleneck, due to the greedy cross traffic.

5.4.4 Experiment 3 - Latency tests with bundling limitation and greedy cross traffic

This experiment has the exact same configuration as experiment 2 (section 5.4.3), except for one property. The bundling rate of the RDB streams in experiment 2 were limited by the DPIFL test, however, when they were allowed to bundle, they would included as much redundant data as possible.

5.4.4.1 Test parameters

The test parameters for this experiment equals that of experiment 2, with one exception:

`rdbsender`

- **Bundling limitation:** 1 previous SKB.

In appendix A.3 the complete test setup can be seen in table A.3, followed by the full set of plots for the experiment.

5.4.4.2 Key results

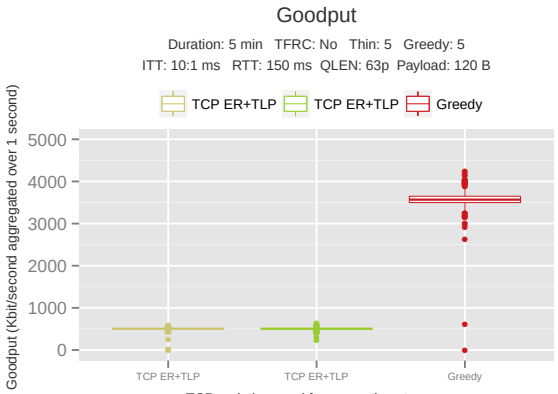
The goal of this experiment is to measure the effect of applying a bundling limit, to the same scenarios tested in experiment 2. We will therefore present some key results from the tests in this experiment, and compare them to the tests in experiment 2, to see how the bundling limitation affects the performance of the RDB streams and the competing streams.

Low ITT scenario (10:1)

We will first look at the goodput plots of the low ITT scenario `E3-5-10:1`, shown in figure 5.17. We see that the ITT values increase slightly from `E3-5-10:1-TCP` to `E3-5-10:1-RDB`. The TCP thin stream ITT in `E3-5-10:1-TCP` average at 19 and 20, whereas in `E3-5-10:1-RDB`, the TCP thin stream have 21 and the RDB streams have 23. While this shows that the RDB streams affect the competing thin stream, the effect is not as severe as observed in the `E2-5-10:1` tests from experiment 2.

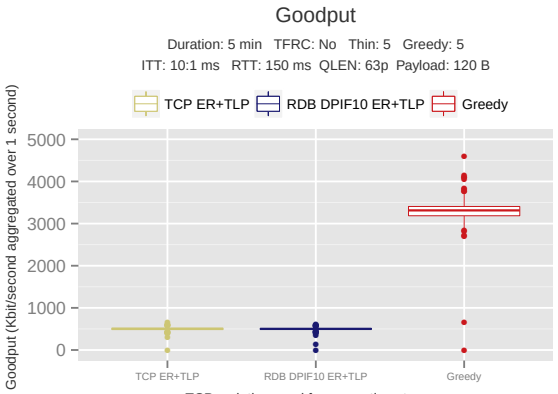
From the throughput plots in figure 5.18 we see that the thin streams send fewer packets in the RDB test, where the TCP thin stream in `E3-5-10:1-TCP` send 79.9K and 74K packets, which, in `E3-5-10:1-RDB`, is reduced to 73.3K for the TCP streams, and 64.3K for the RDB streams.

This shows that the competing TCP thin stream are still affected when competing with RDB, however, the difference compared to the tests in `E2-5-10:1` evident. In the `E3-5-10:1-RDB`, the total packets sent for the thin TCP and RDB streams, is 52.6K and 45.2K, respectively. Comparing the loss rates, we also see



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	18 / 19 / 19	233/238/248	13/1/10/13/16/28
TCP ER+TLP	rdbsender	5	18 / 20 / 22	231/256/284	13/1/9/12/16/28
Greedy	zsender	5	14 / 16 / 17	1446/1447/1447	14/3/10/13/17/51

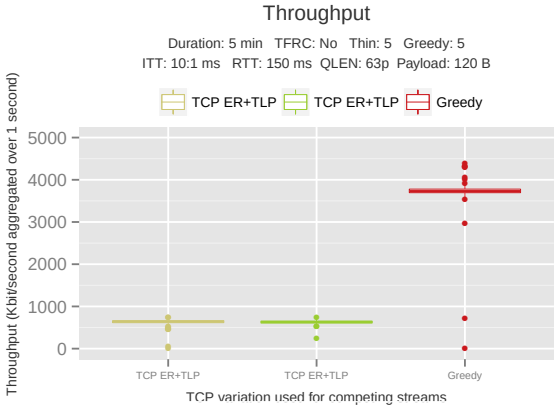
(a) E2-5-10:1-TCP



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	19 / 21 / 22	250/265/283	12/1/9/12/15/29
RDB DPIF10 ER+TLP	rdbsender	5	22 / 23 / 24	450/472/498	12/1/8/11/15/30
Greedy	zsender	5	15 / 17 / 19	1447/1447/1447	14/1/10/13/16/51

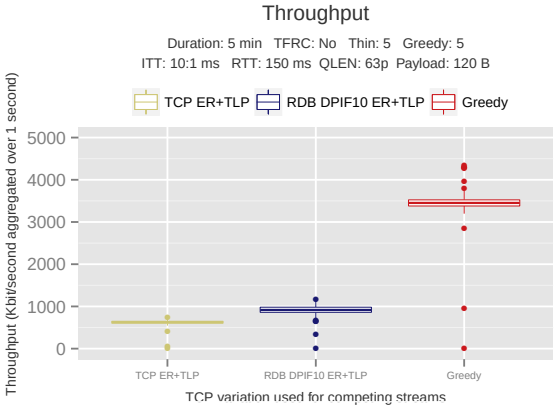
(b) E2-5-10:1-RDB

Figure 5.17



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
79.9 K	1109	1111	0	19.0 M	18.8 M	227.9 K	227.9 K	1.20 / 1.39	11240
74.0 K	1162	1162	0	18.9 M	18.6 M	235.1 K	235.1 K	1.25 / 1.57	10788
94.7 K	1649	1651	0	137.0 M	134.6 M	2.4 M	2.4 M	1.74 / 1.74	13121

(a) E3-5-10:1-TCP



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
72.3 K	1157	1159	0	19.1 M	18.9 M	236.6 K	236.6 K	1.24 / 1.60	10635
64.3 K	1120	399	47.5 K	30.3 M	18.6 M	11.7 M	143.2 K	1.60 / 1.74	3565
87.9 K	1619	1620	0	127.3 M	124.9 M	2.3 M	2.3 M	1.84 / 1.84	12624

(b) E3-5-10:1-RDB

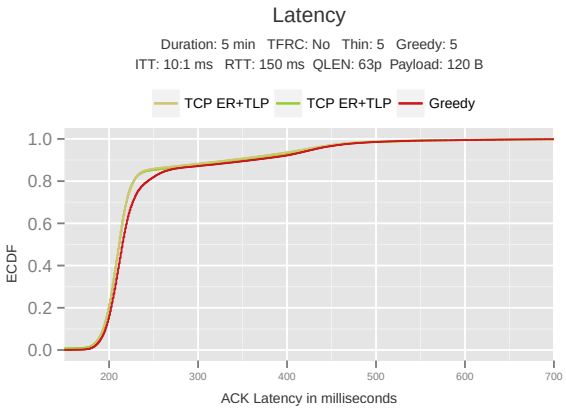
Figure 5.18

that the loss rates for the thin stream of 2.55% and 2.79% E2-5-10:1-RDB, have been reduced to 1.6%, and 1.74% in E3-5-10:1-RDB.

From the latency plot of E3-5-10:1-RDB (subfigure 5.19.(b)) we also see that, while the TCP thin stream achieve higher than the RDB streams, where with TCP average is 246 ms and RDB average is 229 ms, the increase to the latency values is less severe than in E2-5-10:1-RDB, where the average for the TCP streams is 279 ms.

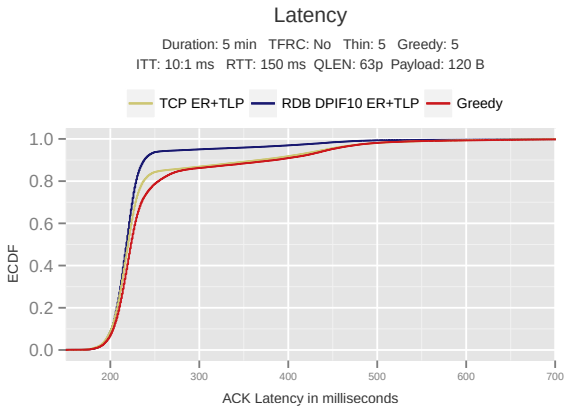
Effects of TFRC

The goodput and throughput plots in figure 5.20 show some interesting results. Looking at the ITT values, we see that they have not changed for the TCP thin



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
236	211	215	220	228	336	390	419	432	447	467	514	721	3347	3420
236	211	215	219	228	350	400	425	438	452	475	542	746	899	999
242	215	220	227	243	361	410	430	441	456	478	530	718	1254	1259

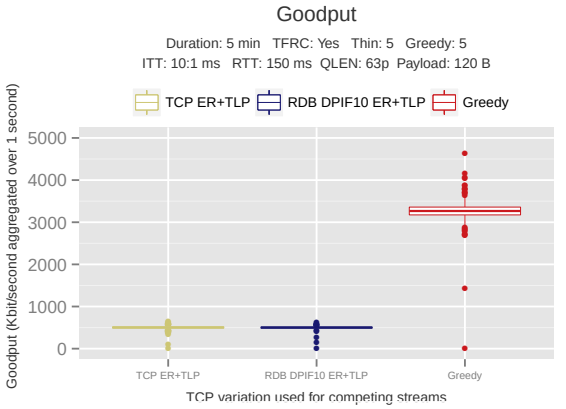
(a) E3-5-10:1-TCP



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	220	224	228	237	367	418	443	456	472	497	557	718	905	1152
229	218	221	225	229	238	246	298	355	402	438	476	773	1019	1707
253	223	228	235	254	385	428	447	460	474	496	554	843	2287	2299

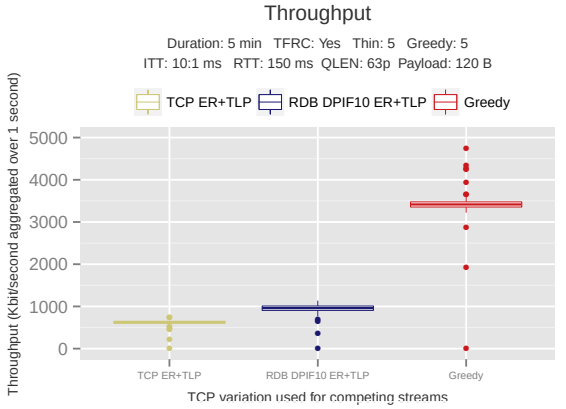
(b) E3-5-10:1-RDB

Figure 5.19



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND	Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
TCP ER+TLP	wsender	5	19 / 21 / 22	247/269/279	12/1/9/12/15/28	71.1 K	1188	1190	0	19.1 M	18.8 M	262.3 K	262.3 K	1.37 / 1.67	10563
RDB DPIF10 ER+TLP	rdbsender	5	17 / 18 / 19	363/381/404	14/1/10/13/18/71	80.7 K	1328	535	54.7 K	30.7 M	18.5 M	12.2 M	122.2 K	1.46 / 1.65	5897
Greedy	zsender	5	16 / 17 / 18	1446/1447/1447	14/3/10/12/16/43	87.1 K	1644	1646	0	126.1 M	123.7 M	2.4 M	2.4 M	1.89 / 1.89	12783

(a) E3-5-10:1-TCP



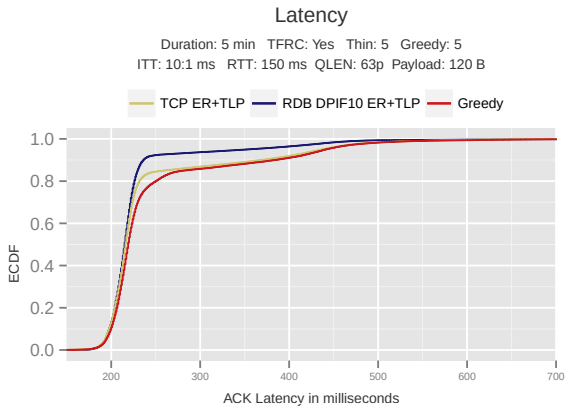
(b) E3-5-10:1-RDB

Figure 5.20

streams compared to E3-5-10:1-RDB, but the ITT values for the RDB streams are lower. Being 17/18/19 in E3-5-10:1-RDB-TFRC, they are even lower than the values for the thin stream in E3-5-10:1-TCP (18/20/22) where only TCP streams compete.

From, the throughput plot E3-5-10:1-RDB-TFRC (subfigure 5.20.(b)) we see that the RDB streams send considerably more packets compared to E3-5-10:1-RDB, 80.7K and 64.4K, respectively.

Looking at the latency plot in subfigure 5.20.(c), we see the same effect as observed in E2-5-10:1-RDB-TFRC from experiment 2, where both the TCP and RDB thin streams achieves a slightly lower average ACK latency when TFRC is enabled for the RDB streams.



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
242	216	220	224	232	368	414	437	449	464	485	542	678	1110	1747
228	215	218	222	226	236	272	353	386	415	440	472	799	1043	1257
248	219	224	230	250	384	423	441	452	467	491	544	722	859	1151

(c) `E3-5-10:1-RDB-TFRC`

Figure 5.20

Differences in comparison to experiment 2

In the `E2-5-10:1` tests in experiment 2, we found that the average ACK latency increased from 236 ms in `E2-5-10:1-TCP` with only TCP streams, to 279 ms in `E2-5-10:1-RDB`, and 274 ms in `E2-5-10:1-RDB-TFRC`, when they compete with RDB streams. The difference to the results for `E3-5-10:1` is significant. From the latency plot of `E3-5-10:1-RDB-TFRC` (subfigure 5.20.(c)), we see that the average ACK latency only increases from 236 ms (in `E3-5-10:1-TCP`) to 246 ms in `E3-5-10:1-RDB`, and 242 ms in `E3-5-10:1-RDB-TFRC`.

It is evident that the bundling limitation enforced on the RDB streams in these tests, contribute greatly to reducing the negative effects inflicted on the competing TCP thin streams.

5.4.4.3 Summary

The results from this experiment shows that the ACK latency can be greatly reduced by only bundling one older data segment with each new packet. Enforcing this limitation on bundling will result in less robustness against events where two packets in a row are lost. However, bundling only one previous segment will still recover most occurrences of sporadic loss. The difference in how the RDB streams affect the competing streams is also very clear. In a congested scenario, the increased payload of the RDB packets will result in higher loss rates, at least if the rate limitation scheme we have used on the experiments is representable for real world scenarios. If, however, the rate limitation imposes an artificial limitation that will not be observable in “the wild”, then the harsh bundling restriction used in this experiment may not be necessary.

5.4.5 Experiment 4 - Fairness experiments

We have performed a set of fairness tests with three hosts, where one host (`zsender`) sends greedy traffic, and two hosts (`wsender` and `rdbsender`) send isochronous streams, which differ from the thin streams in the previous experiments only by the amount of data per `send` call, and the lower ITT rates.

To see the potential of abusing the redundant bundle technique, the `rdbsender` sends 400 byte chunks of data to the kernel at a fast rate. This leaves room to bundle two earlier segments if the stream is able to transmit continuously without send buffering delay on the sender side. If the rate at which the application writes data to the kernel is too great, the TCP engine will fill each the SKBs, leaving no room for redundant data.

As a reference point, the `wsender` host is set to send the same traffic patterns as `rdbsender`, but with the TCP New Reno CC from Linux kernel version 3.15.

5.4.5.1 Test parameters

The test parameters for this experiment is as follows:

`Network`

- **RTT:** 150 ms
- **Rate limit:** 5000kbit
- **Queue:** pfifo with 63 packet limit

`rdbsender`

- **Stream type:** Isochronous
- **Stream count:** 3, 5, 7, 10, and 13
- **Payload:** 400 bytes
- **ITT:** 1, 5:1, 10:1, 15:1, 30:3, and 50:5 ms
- **TCP variation:**
  - **TCP New Reno** from Linux kernel version 3.15 with ER+TLP
  - **RDB** with DPIFL10
  - **RDB** with DPIFL10 + TFRC
  - **RDB** with DPIFL20
  - **RDB** with DPIFL20 + TFRC

`wsender`

- **Stream type:** Isochronous
- **Stream count:** 3, 5, 7, 10, and 13
- **Payload:** 400 bytes
- **ITT:** 1, 5:1, 10:1, 15:1, 30:3, and 50:5 ms
- **TCP variation:** TCP New Reno from Linux kernel version 3.15 with ER+TLP

zsender

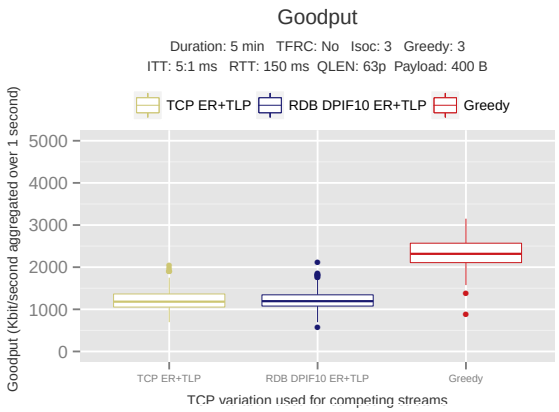
- **Stream type:** Greedy
- **Stream count:** 3, 5, 7, 10, and 13
- **TCP variation:** system default TCP Cubic with ER+TLP.

In appendix A.4 the complete test setup can be seen in table A.4, followed by the full set of plot results for the experiment.

5.4.5.2 Key results

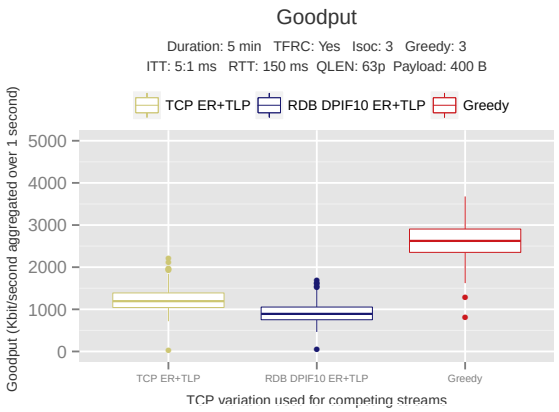
In the tests with the lowest ITT, such as `E4-3-5:1-RDB` (subfigure 5.21.(a)), we can see that the RDB are for the most part unable to bundle. The result is that the average payloads of the the isochronous streams is close to the MSS of 1460 bytes.

In `E4-3-5:1-RDB-TFRC`, the same test with TFRC enabled (subfigure 5.21.(b)), the only difference is that the TFRC CC is stricter, giving the RDB streams a lower CWND.



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	26 / 28 / 30	1387/1421/1444	11/1/8/11/14/36
RDB DPIF10 ER+TLP	rdbsender	3	26 / 28 / 30	1442/1442/1443	11/1/8/11/14/43
Greedy	zsender	3	14 / 14 / 14	1447/1447/1447	20/6/16/19/24/48

(a) `E4-3-5:1-RDB`



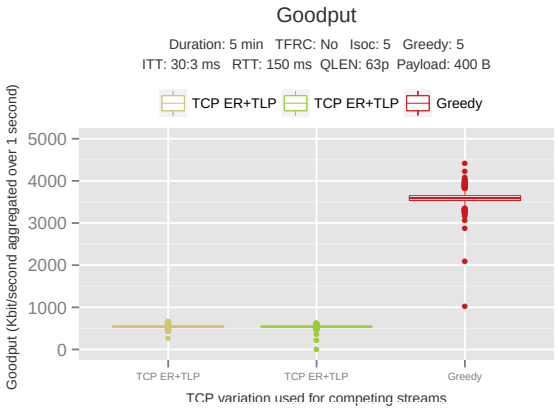
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	26 / 27 / 27	1405/1416/1426	12/1/8/11/14/34
RDB DPIF10 ER+TLP	rdbsender	3	36 / 37 / 37	1418/1432/1443	9/1/6/8/12/33
Greedy	zsender	3	12 / 13 / 13	1446/1446/1447	22/1/17/22/27/46

(b) `E4-3-5:1-RDB-TFRC`

Figure 5.21

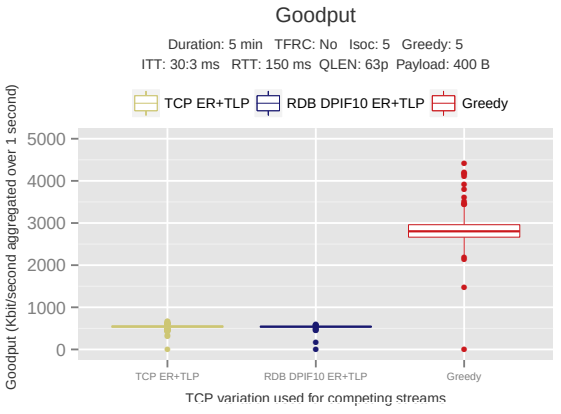
In a higher ITT scenario, such as `E4-5-30:3`, we see from figure 5.22 how the average ITT for both the isochronous streams are slightly higher in the `E4-5-30:3-RDB`. From the throughput plots in figure 5.23, we see that while being able to bundle on most packets, the RDB streams only achieve half the throughput rate of the greedy streams. The goodput is still not higher than the isochronous TCP streams.

Moving to the `E4-10-30:3` tests, we see a congested scenario where the RDB streams are allowed to bundle on more than half of the packets sent. Figure 5.24 shows the throughput plots of the two tests `E4-10-30:3-RDB` and `E4-10-30:3-RDB-TFRC`. In the non-TFRC test, the RDB streams achieve considerably less throughput than the greedy streams. In the TFRC test, they get closer, but is still not able to match the throughput of the greedy streams. The goodput plot in figure 5.25 shows that the goodput is at the same level of the isochronous TCP streams, leaving nothing to gain from using RDB.



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 32	426/432/440	11/1/8/12/13/14
TCP ER+TLP	rdbsender	5	31 / 32 / 32	426/432/437	11/1/8/12/13/14
Greedy	zsender	5	15 / 16 / 16	1446/1447/1447	16/4/12/15/18/42

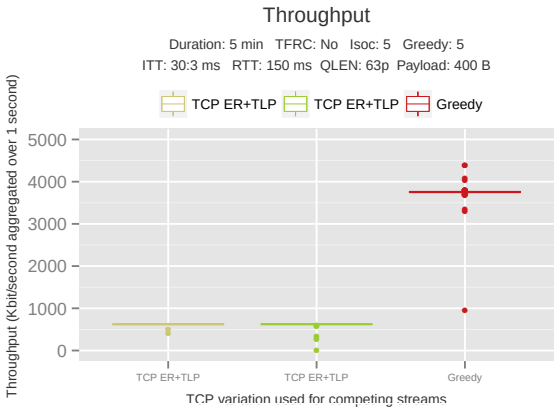
(a) `E4-5-30:3-TCP`



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	32 / 32 / 32	434/439/444	11/1/8/11/13/14
RDB DPIF10 ER+TLP	rdbsender	5	33 / 34 / 34	1157/1160/1166	10/1/8/10/13/14
Greedy	zsender	5	18 / 20 / 22	1446/1447/1447	13/4/10/12/15/37

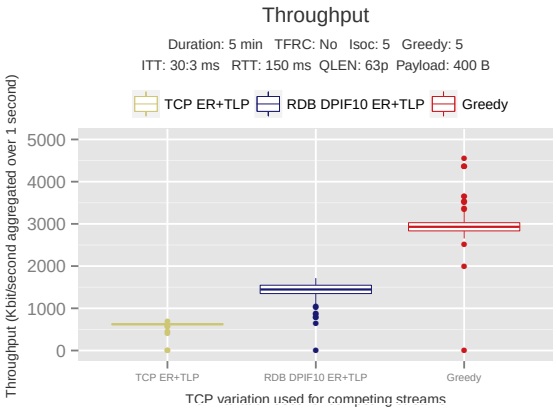
(b) `E4-5-30:3-RDB`

Figure 5.22



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
47.6 K	540	540	0	20.6 M	20.3 M	220 K	220 K	1.07 / 1.14	4329
46.8 K	534	536	0	20.2 M	20.0 M	220 K	220 K	1.08 / 1.14	4276
94.7 K	1164	1164	0	137 M	135 M	1.7 M	1.7 M	1.23 / 1.23	12427

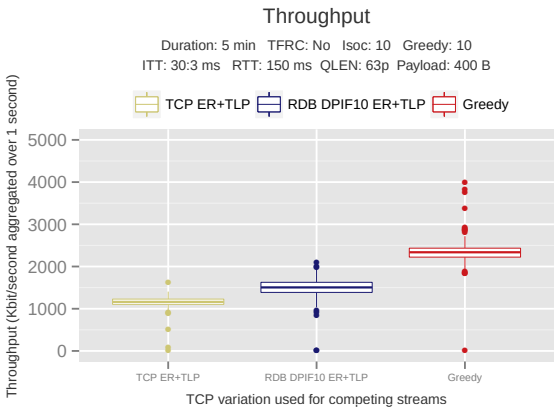
(a) E4-5-30:3-TCP



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
46.8 K	589	591	0	20.6 M	20.3 M	242 K	242 K	1.18 / 1.26	4672
43.8 K	597	33	40.1 K	50.9 M	20.0 M	30.8 M	24.8 K	1.38 / 1.36	181
75.4 K	1439	1441	0	109 M	107 M	2.1 M	2.1 M	1.91 / 1.91	11714

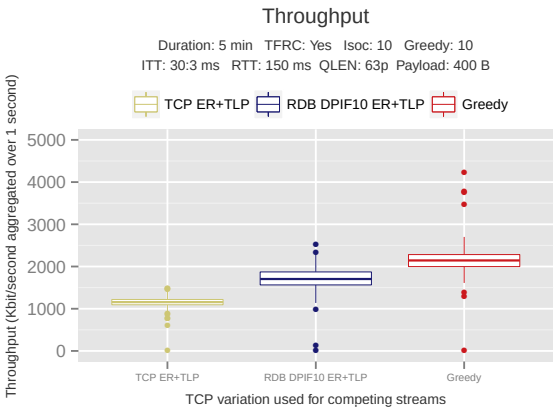
(b) E4-5-30:3-RDB

Figure 5.23



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
50.7 K	2781	2788	0	42.8 M	40.6 M	2.1 M	2.1 M	4.94 / 5.48	11685
47.6 K	2621	1476	20.7 K	55.9 M	39.9 M	16.1 M	1.7 M	5.48 / 5.50	5023
62.2 K	3511	3518	0	89.9 M	84.9 M	5.1 M	5.1 M	5.64 / 5.64	14395

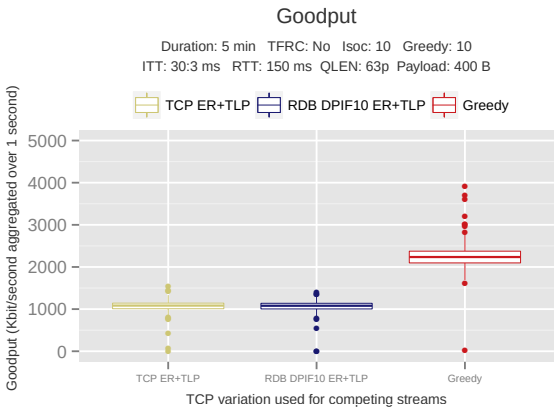
(a) E4-10-30:3-RDB



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
50.0 K	2747	2756	0	42.6 M	40.4 M	2.1 M	2.1 M	4.97 / 5.49	11459
55.0 K	3302	1518	31.2 K	63.8 M	40.0 M	23.8 M	1.7 M	5.98 / 6.00	5404
57.2 K	3510	3516	0	82.8 M	77.7 M	5.1 M	5.1 M	6.13 / 6.13	13704

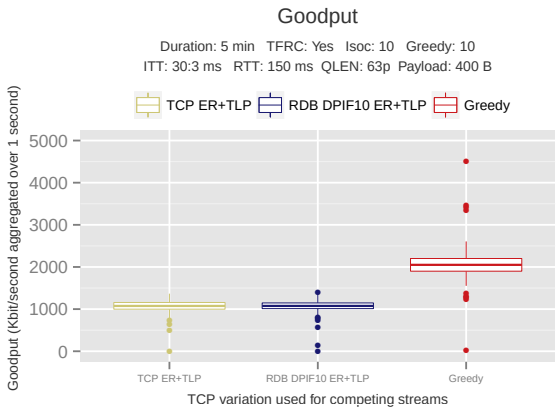
(b) E4-10-30:3-RDB-TFRC

Figure 5.24



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	53 / 59 / 62	745/845/893	6/1/4/5/7/14
RDB DPIF10 ER+TLP	rdbsender	10	58 / 62 / 69	1150/1175/1205	6/1/4/5/7/14
Greedy	zsender	10	44 / 49 / 58	1445/1446/1447	6/1/5/6/7/26

(a) E4-10-30:3-RDB



Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	58 / 60 / 62	811/851/893	6/1/4/5/7/14
RDB DPIF10 ER+TLP	rdbsender	10	52 / 54 / 56	1139/1161/1176	6/1/5/6/8/47
Greedy	zsender	10	46 / 53 / 56	1446/1446/1447	6/1/4/5/7/27

(b) E4-10-30:3-RDB-TFRC

Figure 5.25

5.4.5.3 Summary

The main goal of these experiments is to show that RDBv2 maintains TCP-friendliness, even in scenarios designed for optimizing the possibility of gaining an unfair advantage.

As the results show, in none of the tests do the RDB streams get an advantage over the other streams throughput-wise.

Making the abuser application smarter

When the TCP output queue is increasing we know that the effect is that RDB is disabled. A smarter abuser could use this knowledge to try to improve the chance of having more data bundled in RDB packets. The amount the queued outgoing data on the socket can be fetched from the kernel (`SIOCOUTQ` on Linux), and based on this the sender application could adjust the send rate to avoid filling the queue. The ideal would be to adjust the pace of the `send` calls to the kernel, such that the there is just enough room to bundle redundant data with each packet.

5.5 Summary

In this chapter, we have presented our test setup, and described the key findings in each experiment.

The experiments show that RDB gives thin stream over TCP significant less delay, mostly by reducing the negative effects of the HOL blocking phenomenon. By imposing limitations on how much redundant data can be bundled, the RDB mechanism can achieve great improvements to the latencies, without affecting competing thin stream significantly.

The range of environments tested in the experiments does not represent the environments where we believe RDB should be used. However, to find which scenarios RDB can be used we also tested different edge case scenarios.

While the lowest ITTs used in the experiments were as low as 10 ms, it does not mean we will argue that it is a reasonable limit for when to identify a stream as thin. However, as the experiments show, RDB streams with such low ITTs can be made to have little affect on competing streams.

Based on the results of the experiments, we argue that RDB definitely can be used by thin streams in many scenarios, without penalizing competing traffic in an unfair manner. However, further tests are required to figure out when and where to enable RDB.

Chapter 6

Conclusion

6.1 Summary

One often hears about how fast things change, especially in the realm of computer science. While being first specified in the "ancient" times of 1970s (*RFC675*), TCP is still the first choice for most general applications on the Internet. It's ease of use and reliability guarantees makes it a solid choice of protocol for applications that do not transmit time-dependent data. While there are alternatives to TCP, it would be of great benefit to improve TCP's performance for applications with strict latency requirements.

The RDB mechanism we have developed and tested, does not aim to replace the default CCs used today, but to be an easy alternative for applications that have strict latency requirements, and for various reasons need to use TCP.

In the later years, research has started focusing on the latency performance of TCP, with new mechanisms emerging, like LT, mFR, ER and TLP. What all these mechanisms have in common is that they try to minimize the time before lost packets are retransmitted by the TCP engine. The problem is that they wait until they believe it is likely that a loss has occurred. This means that the per-packet latencies for lost packets are increased by a minimum of one RTT, but often it can be more.

This is where RDB takes a different path, and retransmits un-ACKed data segments simply because it *may* be(come) lost. By bundling old un-ACKed data that probably will be redundant, it will proactively ensure that the per-packet (or per-segment) latency is not drastically reduced in case of loss. We argue that this can be justified because RDB does not create extra packets on the wire by ensuring to create packets that are below the MSS limit.

The TFRC CC we have implemented in RDBv2 raises the CWND for thin streams compared to the latest version of the Linux kernel version 3.19, which causes more packets on the wire in some cases. This is not because it allows the stream to send more data, but because, the data is, to a greater degree, queued on the sender side, when the CWND is a limiting factor.

Through experiments we have confirmed the vast improvements in latencies reported by *Petlund* [2009, a4] when using RDB. We have also verified that the RDBv2 implementation is TCP fair, at least within the scope of our test environ-

ment.

6.2 Contributions

The work in this thesis has focused on how to improve the latency for thin streams using TCP. While some recent work has focused on mechanisms for reducing the retransmission delay for thin streams over TCP (section 2.6), the lost data segments still require retransmissions. The RDB mechanism modifies the packets sent by TCP by bundling already sent data with the packets containing new, untransmitted data. By bundling redundant data with the data packets, the stream can recover from sporadic packet loss without having to wait for a retransmission from the sender host.

Based on earlier work on RDB (*Evensen* [2008, a5]; *Petlund* [2009, a4]), this thesis tries to tame the RDB mechanism, to ensure that the latency improvements for the streams utilizing RDB is balanced against the negative effects inflicted on competing traffic.

The main contributions of this thesis can be summarized as follows:

- We have evaluated the thin-stream classification mechanism in the Linux kernel. Further, we have developed an alternative method, which uses the RTT, together with the user space application's transmission patterns, to identify if the stream is thin.
- We have identified an issue with recent changes to the Linux kernel's TCP engine that severely affects the latency for thin streams negatively.
- Implementation of RDB as a Linux kernel module, that enables a sender host to transmit data to a TCP receiver using the RDB technique.
- To help thin streams maintain a more stable throughput rate than provided by standard TCP, we have implemented a CC in the kernel module which utilizes the equation based throughput calculation for TFRC-SP.
- Investigated and developed methods for detecting packet loss that RDB hides from the TCP engine.

The main contributions from the experiments performed on RDB, can be summarized as follows:

- Investigating how the redundant bundling mechanism performs under different test scenarios, and finding a tradeoff between improving the latency and increasing the payload size when bundling redundant data.
- The latency gains achieved with different bundling rates, i.e., limitations to how much data can be bundled with each packet.
- How RDB streams affect competing traffic under different test scenarios.
- How RDB can be prevented from being abused to obtain an advantage over competing traffic.

6.3 Future work

Thin streams over TCP have been overlooked for a long time, and require further attention to be able to compete with other transport protocols. Some candidates for future work is outlined next:

Work specifically on RDB.

- **Investigate the recent changes to the Linux kernel**
  Recent changes to the Linux kernel's TCP engine seem to affect the latency of thin streams (described in section 3.4. The negative effects these changes have, may force current applications sending thin streams over TCP over on other protocols. This issue should be investigated further, firstly by finding out if the behavior is intended or not. Hopefully it can be regarded as a bug, so that work on how to fix and improve the situation can be done. If it is intended, the standards need to be changed if thin streams should have a future together with TCP.
- **Improving the loss detection mechanisms for RDB.**
  Of the techniques we presented in section 4.2 for detecting loss, only the technique looking at multiple ACKed packets were implemented in RDBv2. As this technique is reliable as long as there is no packet reordering or loss of ACK packets, it may perform worse in the "real world". However, the lack of mechanisms for detecting reordering can only lead to an over-estimation of the loss rate, which does not lead to unfairness in favor of the RDB streams. Further improvements to the loss detection can be made by implementing the technique detailed in section 4.2.0.4, where the DSACK information in the ACKs is utilized to identify which packet was lost.
- **Performance profiling**
  To measure the resource overhead caused by the different parts of the module it is necessary to perform profiling.
- **Measure the true Application layer latency**
  Due to the limitations of the ACK latency measurement explained in section 5.1.1.2, further work should be done on finding a solution on how to measure the application layer latency. This is important to get more precise latency measurements, allowing a fair comparison between the results of mechanisms and takes into account the sender side queuing delay.
    - One possible solution is to use kprobes to hook into the relevant functions in the kernel to get the timestamps of when the the payload enters and leaves the kernel.
    - Another solution could be to implement this in the sender application such as streamzero, that saves the timestamps for the payload. By using the same computer for sending and receiving, the clock drift issue can be evaded.

Bibliography

References (a)

[a1] *Carsten Griwodz* and *Pål Halvorsen*. "The fun of using TCP for an MMORPG". In: *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (May 2006), pp. 1–7 (cit. on pp. i, 10, 11, 13).

[a2] *Bing Wang* et al. "Multimedia Streaming via TCP: An Analytic Performance Study". In: *ACM Trans. Multimedia Comput. Commun. Appl.* 4.2 (May 2008), 16:1–16:22. ISSN: 1551-6857. DOI: 10.1145/1352012.1352020 (cit. on p. i).

[a3] *Ian Mcdonald* and *Richard Nelson*. "Congestion control advancements in Linux". In: *Linux Conf AU*. Jan. 2006 (cit. on pp. xxii, 16, 82).

[a4] *Andreas Petlund*. "Improving latency for interactive, thin-stream applications over reliable transport". PhD thesis. University of Oslo, Dec. 2009 (cit. on pp. xxiii, xxiv, 11, 12, 30, 133, 134).

[a5] *Kristian Evensen*. "Improving TCP for time-dependent applications". MA. Thesis. University of Oslo, May 2008 (cit. on pp. xxiii–xxv, 3, 38, 134).

[a6] *Wolfgang John* and *Sven Tafvelin*. "Analysis of Internet Backbone Traffic and Header Anomalies Observed". In: *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*. IMC '07. San Diego, California, USA: ACM, 2007, pp. 111–116. ISBN: 978-1-59593-908-1 (cit. on pp. 1, 7).

[a7] *Saleem Bhatti* et al. "Revisiting inter-flow fairness". In: *Proceedings of the 5th International Conference on Broadband Communications, Networks, and Systems (BROADNETS)*. London: IEEE, Sept. 2008, pp. 585–592 (cit. on pp. 1, 16, 28).

[a8] *Deepak Bansal* and *Hari Balakrishnan*. "Binomial Congestion Control Algorithms". In: *INFOCOM*. 2001 (cit. on pp. 1, 16, 21, 22).

[a9] *Lawrence Stewart* et al. "Multimedia-unfriendly TCP Congestion Control and Home Gateway Queue Management". In: *Proceedings of the Second Annual ACM Conference on Multimedia Systems*. MMSys '11. San Jose, CA, USA: ACM, 2011, pp. 35–44. ISBN: 978-1-4503-0518-1 (cit. on pp. 1, 30, 56).

[a10] *DongJin Lee*, *Brian Carpenter*, and *Nevil Brownlee*. "Media Streaming Observations: Trends in UDP to TCP Ratio". In: *International Journal on Advances in Systems and Measurements* 3.3-4 (Apr. 2010), pp. 147–162 (cit. on pp. 2, 9).

[a11] *D. E. Comer* et al. "Computing As a Discipline". In: *Commun. ACM* 32.1 (Jan. 1989). Ed. by *Peter J. Denning*, pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239 (cit. on p. 3).

[a12] *Eli Brosh* et al. "The Delay-friendliness of TCP for Real-time Traffic". In: *IEEE/ACM Trans. Netw.* 18.5 (Oct. 2010), pp. 1478–1491. ISSN: 1063-6692 (cit. on pp. 7, 8).

[a13] *Lei Guo* et al. "Delving into Internet Streaming Media Delivery: A Quality and Resource Utilization Perspective". In: *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. IMC '06. Rio de Janeriro, Brazil: ACM, 2006, pp. 217–230. ISBN: 1-59593-561-4 (cit. on pp. 8, 10).

[a14] *Florin Dobrian* et al. "Understanding the Impact of Video Quality on User Engagement". In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2011), pp. 362–373. ISSN: 0146-4833 (cit. on p. 9).

[a15] *Mark Claypool* and *Kajal Claypool*. "Latency and player actions in online games." In: *Communications of the ACM* 49.11 (Nov. 2005), pp. 40–45 (cit. on p. 10).

[a16] *Peter Quax* et al. "Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game". In: *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames '04. Portland, Oregon, USA, 2004, pp. 152–156. ISBN: 1-58113-942-X (cit. on p. 10).

[a17] *Tanja Lang*, *Philip Branch*, and *Grenville Armitage*. "A Synthetic Traffic Model for Quake3". In: *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*. ACE '04. Singapore, 2004, pp. 233–238. ISBN: 1-58113-882-2 (cit. on p. 13).

[a18] *Markus Simon Fuchs*. "Time-Dependent Thin Transport Layer Streams: Characterization, Empirical Observation and Protocol Support". MA. Thesis. University of Kaiserslautern, Jan. 2014 (cit. on pp. 13, 28).

[a19] *Nandita Dukkipati*, *Matt Mathis*, et al. "Proportional Rate Reduction for TCP". In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*. IMC '11. Berlin, Germany, 2011, pp. 155–170 (cit. on pp. 14, 16, 20, 25, 32).

[a20] *Matthew Mathis* and *Jamshid Mahdavi*. "Forward Acknowledgement: Refining TCP Congestion Control". In: *SIGCOMM Comput. Commun. Rev.* 26.4 (Aug. 1996), pp. 281–291. ISSN: 0146-4833. DOI: 10.1145/248157.248181 (cit. on p. 16).

[a21] *V. Jacobson*. "Congestion Avoidance and Control". In: *SIGCOMM Comput. Commun. Rev.* 18.4 (Aug. 1988), pp. 314–329. ISSN: 0146-4833 (cit. on pp. 16, 17, 23, 30).

[a22] *Sally Floyd* and *Kevin Fall*. "Promoting the Use of End-to-end Congestion Control in the Internet". In: *IEEE/ACM Trans. Netw.* 7.4 (Aug. 1999), pp. 458–472. ISSN: 1063-6692 (cit. on pp. 16, 29).

[a23] *Sangtae Ha*, *Injong Rhee*, and *Lisong Xu*. "CUBIC: A New TCP-friendly High-speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105 (cit. on p. 19).

[a24] *Claudio Casetti* et al. "TCP Westwood: End-to-end Congestion Control for Wired/Wireless Networks". In: *Wirel. Netw.* 8.5 (Sept. 2002), pp. 467–479. ISSN: 1022-0038 (cit. on p. 19).

[a25] *Sally Floyd* et al. "Equation-based Congestion Control for Unicast Applications". In: *SIGCOMM Comput. Commun. Rev.* 30.4 (Aug. 2000), pp. 43–56. ISSN: 0146-4833 (cit. on p. 20).

[a26] *Jitendra Padhye* et al. "Modeling TCP Throughput: A Simple Model and Its Empirical Validation". In: *SIGCOMM Comput. Commun. Rev.* 28.4 (Oct. 1998), pp. 303–314. ISSN: 0146-4833. DOI: 10.1145/285243.285291 (cit. on p. 21).

[a27] *P. Karn* and *C. Partridge*. "Innovations in Internetworking". In: ed. by *C. Partridge*. Norwood, MA, USA: Artech House, Inc., 1988. Chap. Improving Round-trip Time Estimates in Reliable Transport Protocols, pp. 266–271. ISBN: 0-89006-337-0 (cit. on p. 24).

[a28] *Pasi Sarolahti* and *Alexey Kuznetsov*. "Congestion Control in Linux TCP". In: *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 49–62. ISBN: 1-880446-01-4 (cit. on p. 27).

[a29] *Byung-Jae Kwak*, *Nah-Oak Song*, and *Leonard E. Miller*. "Performance Analysis of Exponential Backoff". In: *IEEE/ACM Trans. Netw.* 13.2 (Apr. 2005), pp. 343–355. ISSN: 1063-6692 (cit. on p. 27).

[a30] *Amit Mondal* and *Aleksandar Kuzmanovic*. "Removing Exponential Backoff from TCP". In: *SIGCOMM Comput. Commun. Rev.* 38.5 (Sept. 2008), pp. 17–28. ISSN: 0146-4833 (cit. on p. 27).

[a31] *Bob Briscoe*. "Flow Rate Fairness: Dismantling a Religion". In: *SIGCOMM Comput. Commun. Rev.* 37.2 (Mar. 2007), pp. 63–74. ISSN: 0146-4833 (cit. on p. 30).

[a32] *Andreas Petlund*. "TCP Thin-Stream Modifications: Reduced Latency for Interactive Applications." In: *Linux Journal* 2012.219 (July 2012), pp. 82–91 (cit. on p. 32).

[a33] *Hari Balakrishnan* et al. "TCP Behavior of a Busy Internet Server: Analysis and Improvements". In: *Proceedings of IEEE INFOCOM*. Vol. 1. San Francisco, CA, Mar. 1998, pp. 252–262 (cit. on p. 32).

[a34] *Jonas Markussen*. "Experimentally finding the right aggressiveness for retransmissions in thin TCP streams". MA. Thesis. University of Oslo, Aug. 2014 (cit. on pp. 38, 94).

[a35] *Espen Søgård Paaby*. "Evaluation of TCP retransmission delays". MA. Thesis. University of Oslo, May 2006 (cit. on p. 38).

[a36] *Nandita Dukkipati*, *Tiziana Refice*, et al. "An Argument for Increasing TCP's Initial Congestion Window". In: *SIGCOMM Comput. Commun. Rev.* 40.3 (June 2010), pp. 26–33. ISSN: 0146-4833 (cit. on p. 55).

[a37] *Wai-tian Tan* and *Avideh Zakhor*. "Real-Time Internet Video Using Error Resilient Scalable Compression and TCP-Friendly Transport Protocol". In: *IEEE Transactions on Multimedia* 1.2 (June 1999), pp. 172–186 (cit. on p. 56).

[a38] *R. Ramaswamy*, *Ning Weng*, and *T. Wolf*. "Characterizing network processing delay". In: *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*. Vol. 3. Nov. 2004, 1629–1634 Vol.3. DOI: 10.1109/GLOCOM.2004.1378257 (cit. on pp. 87, 89).

[a39] *Jim Gray*. "The Cost of Messages". In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. PODC '88. Toronto, Ontario, Canada: ACM, 1988, pp. 1–7. ISBN: 0-89791-277-2. DOI: 10.1145/62546.62547 (cit. on p. 88).

[a40] *Jim Gray*. "Rules of Thumb in Data Engineering". In: *Proceedings of the 16th International Conference on Data Engineering*. ICDE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 3–. ISBN: 0-7695-0506-6 (cit. on p. 88).

[a41] *A. P. Foong* et al. "TCP Performance Re-visited". In: *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 70–79. ISBN: 0-7803-7756-7 (cit. on p. 89).

[a42] *George F. Riley* and *Thomas R. Henderson*. "The ns-3 Network Simulator". English. In: *Modeling and Tools for Network Simulation*. Ed. by *Klaus Wehrle*, *Mesut Güneş*, and *James Gross*. Springer Berlin Heidelberg, 2010, pp. 15–34. ISBN: 978-3-642-12330-6. DOI: 10.1007/978-3-642-12331-3_2 (cit. on p. 91).

[a43] *Mats Rosbach*. "Verification of Network Simulators". MA. Thesis. University of Oslo, Nov. 2012 (cit. on p. 91).

[a44] *Kathleen Nichols* and *Van Jacobson*. "Controlling Queue Delay". In: *Queue* 10.5 (May 2012), 20:20–20:34. ISSN: 1542-7730. DOI: 10.1145/2208917.2209336 (cit. on p. 93).

[a45] *G. Vu-Brugier* et al. "A Critique of Recently Proposed Buffer-sizing Strategies". In: *SIGCOMM Comput. Commun. Rev.* 37.1 (Jan. 2007), pp. 43–48. ISSN: 0146-4833. DOI: 10.1145/1198255.1198262 (cit. on p. 93).

Online References (b)

[b1] *Andreas Petlund. TCP thin linear timeouts*. 2010.
URL: http://git.kernel.org/linus/36e31b (visited on 11/2014) (cit. on pp. xxiii, 31).

[b2] *Andreas Petlund. TCP thin dupack*. 2010.
URL: http://git.kernel.org/linus/7e3801 (visited on 11/2014) (cit. on pp. xxiv, xxvi, 33).

[b3] *Nandita Dukkipati. Tail Loss Probe (TLP)*. 2013.
URL: http://git.kernel.org/linus/6ba8a3 (visited on 11/2014) (cit. on pp. xxv, 35).

[b4] *Internetworldstats.com. INTERNET GROWTH STATISTICS*.
URL: http://www.internetworldstats.com (visited on 12/2014) (cit. on p. 1).

[b5] *Cisco. Cisco Visual Networking Index: Forecast and Methodology, 2013-2018*. Tech. rep. Cisco, June 2014.
URL: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html (visited on 10/2014) (cit. on pp. 8–10).

[b6] *TeleGeography Report*. 2013.
URL: http://www.telegeography.com/research-services/telegeography-report-database (visited on 11/2014)
ARCHIVED URL: http://web.archive.org/web/20131718353200/http://www.telegeography.com/research-services/telegeography-report-database/ (cit. on p. 8).

[b7] *International Telecommunication Union (ITU-T). One-way Transmission Time, ITU-T Recommendation G.114*. 2003.
URL: https://www.itu.int/rec/T-REC-G.114-200305-I (visited on 06/2014) (cit. on pp. 8, 9).

[b8] *FreeBSD 9 source code*.
VERSION: svn checkout -r252825. (sys/netinet/tcp_timer.h line 111).
URL: http://svnweb.freebsd.org/base/stable/9/sys/netinet/tcp_timer.h?revision=252825&view=markup#l111 (visited on 06/2014) (cit. on p. 27).

[b9] *Yuchung Cheng. Early Retransmit (ER)*. 2012.
URL: http://git.kernel.org/linus/eed530 (visited on 11/2014) (cit. on p. 35).

[b10] *Linux kernel IPv4 variables*.
URL: https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt (visited on 11/2014) (cit. on p. 35).

[b11] *Bufferbloat-project. Best Practices for Benchmarking CoDel and FQ CoDel*. Sept. 2014.
URL: https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel (visited on 10/2014) (cit. on pp. 91, 92).

[b12] *Ilpo Järvinen. Comments on RDB prototype from Ilpo Järvinen*. 2009.
URL: https://lkml.org/lkml/2009/1/12/228 (visited on 06/2014) (cit. on pp. 309, 310).

Internet Standards and Drafts (c)

[c1] *S. Floyd* et al. *TCP Friendly Rate Control (TFRC): Protocol Specification*. RFC 5348 (Proposed Standard). Internet Engineering Task Force, Sept. 2008.
URL: http://www.ietf.org/rfc/rfc5348.txt (cit. on pp. xvii, 20, 69–72).

[c2] *S. Floyd* and *E. Kohler*. *TCP Friendly Rate Control (TFRC): The Small-Packet (SP) Variant*. RFC 4828 (Experimental). Internet Engineering Task Force, Apr. 2007.
URL: http://www.ietf.org/rfc/rfc4828.txt (cit. on pp. xvii, 71, 72, 76).

[c3] *R. Ludwig* and *M. Meyer*. *The Eifel Detection Algorithm for TCP*. RFC 3522 (Experimental). Internet Engineering Task Force, Apr. 2003.
URL: http://www.ietf.org/rfc/rfc3522.txt (cit. on pp. xxii, 24, 25).

[c4] *M. Allman*, *V. Paxson*, and *E. Blanton*. *TCP Congestion Control*. RFC 5681 (Draft Standard). Internet Engineering Task Force, Sept. 2009.
URL: http://www.ietf.org/rfc/rfc5681.txt (cit. on pp. xxii, 13, 17, 19).

[c5] *D. Borman* et al. *TCP Extensions for High Performance*. RFC 7323 (Proposed Standard). Internet Engineering Task Force, Sept. 2014.
URL: http://www.ietf.org/rfc/rfc7323.txt (cit. on pp. xxvi, 24–26, 63).

[c6] *J. Postel*. *Transmission Control Protocol*. RFC 793 (INTERNET STANDARD). Updated by RFCs 1122, 3168, 6093, 6528. Internet Engineering Task Force, Sept. 1981.
URL: http://www.ietf.org/rfc/rfc793.txt (cit. on p. 1).

[c7] *L. Coene*. *Stream Control Transmission Protocol Applicability Statement*. RFC 3257 (Informational). Internet Engineering Task Force, Apr. 2002.
URL: http://www.ietf.org/rfc/rfc3257.txt (cit. on p. 2).

[c8] *H. Schulzrinne* et al. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550 (INTERNET STANDARD). Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164. Internet Engineering Task Force, July 2003.
URL: http://www.ietf.org/rfc/rfc3550.txt (cit. on p. 8).

[c9] *C. Perkins*. *RTP and the Datagram Congestion Control Protocol (DCCP)*. RFC 5762 (Proposed Standard). Updated by RFC 6773. Internet Engineering Task Force, Apr. 2010.
URL: http://www.ietf.org/rfc/rfc5762.txt (cit. on p. 8).

[c10] *M. Handley*, *J. Padhye*, and *S. Floyd*. *TCP Congestion Window Validation*. RFC 2861 (Experimental). Internet Engineering Task Force, June 2000.
URL: http://www.ietf.org/rfc/rfc2861.txt (cit. on p. 11).

[c11] *C. Hornig*. *A Standard for the Transmission of IP Datagrams over Ethernet Networks*. RFC 894 (INTERNET STANDARD). Internet Engineering Task Force, Apr. 1984.
URL: http://www.ietf.org/rfc/rfc894.txt (cit. on p. 12).

[c12] *M. Mathis*, *J. Mahdavi*, et al. *TCP Selective Acknowledgment Options*. RFC 2018 (Proposed Standard). Internet Engineering Task Force, Oct. 1996.
URL: http://www.ietf.org/rfc/rfc2018.txt (cit. on p. 15).

[c13] *S. Floyd*, *J. Mahdavi*, et al. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC 2883 (Proposed Standard). Internet Engineering Task Force, July 2000.
URL: http://www.ietf.org/rfc/rfc2883.txt (cit. on p. 16).

[c14] *W. Stevens*. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001 (Proposed Standard). Obsoleted by RFC 2581. Internet Engineering Task Force, Jan. 1997.
URL: http://www.ietf.org/rfc/rfc2001.txt (cit. on p. 19).

[c15] *E. Blanton* et al. *A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP*. RFC 3517 (Proposed Standard). Obsoleted by RFC 6675. Internet Engineering Task Force, Apr. 2003.
URL: http://www.ietf.org/rfc/rfc3517.txt (cit. on p. 20).

[c16] *J. Nagle*. *Congestion Control in IP/TCP Internetworks*. RFC 896. Internet Engineering Task Force, Jan. 1984.
URL: http://www.ietf.org/rfc/rfc896.txt (cit. on p. 22).

[c17] *Greg Minshall*. *A Proposed Modification to Nagle's Algorithm*. June 1999.
URL: http://tools.ietf.org/html/draft-minshall-nagle-01 (visited on 12/2014) (cit. on p. 23).

[c18] *R. Braden*. *Requirements for Internet Hosts - Communication Layers*. RFC 1122 (INTERNET STANDARD). Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864. Internet Engineering Task Force, Oct. 1989.
URL: http://www.ietf.org/rfc/rfc1122.txt (cit. on pp. 23, 26, 27).

[c19] *V. Paxson*, *M. Allman*, et al. *Computing TCP's Retransmission Timer*. RFC 6298 (Proposed Standard). Internet Engineering Task Force, June 2011.
URL: http://www.ietf.org/rfc/rfc6298.txt (cit. on pp. 23, 24, 35).

[c20] *V. Paxson* and *M. Allman*. *Computing TCP's Retransmission Timer*. RFC 2988 (Proposed Standard). Obsoleted by RFC 6298. Internet Engineering Task Force, Nov. 2000.
URL: http://www.ietf.org/rfc/rfc2988.txt (cit. on pp. 26, 27).

[c21] *S. Floyd*. *Metrics for the Evaluation of Congestion Control Mechanisms*. RFC 5166 (Informational). Internet Engineering Task Force, Mar. 2008.
URL: http://www.ietf.org/rfc/rfc5166.txt (cit. on p. 28).

[c22] *M. Allman*, *K. Avrachenkov*, et al. *Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)*. RFC 5827 (Experimental). Internet Engineering Task Force, May 2010.
URL: http://www.ietf.org/rfc/rfc5827.txt (cit. on p. 35).

[c23] *Nandita Dukkipati*, *Neal Cardwell*, et al. *TCP Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses*. July 2012.
URL: http://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01 (visited on 11/2014) (cit. on p. 35).

[c24] *Per Hurtig* et al. *TCP and SCTP RTO Restart*. Oct. 2014.
URL: https://tools.ietf.org/html/draft-ietf-tcpm-rtorestart-04 (visited on 12/2014) (cit. on p. 35).

[c25] *V. Cerf*, *Y. Dalal*, and *C. Sunshine*. *Specification of Internet Transmission Control Program*. RFC 675. Internet Engineering Task Force, Dec. 1974.
URL: http://www.ietf.org/rfc/rfc675.txt (cit. on p. 133).

Appendix A

Experiments results

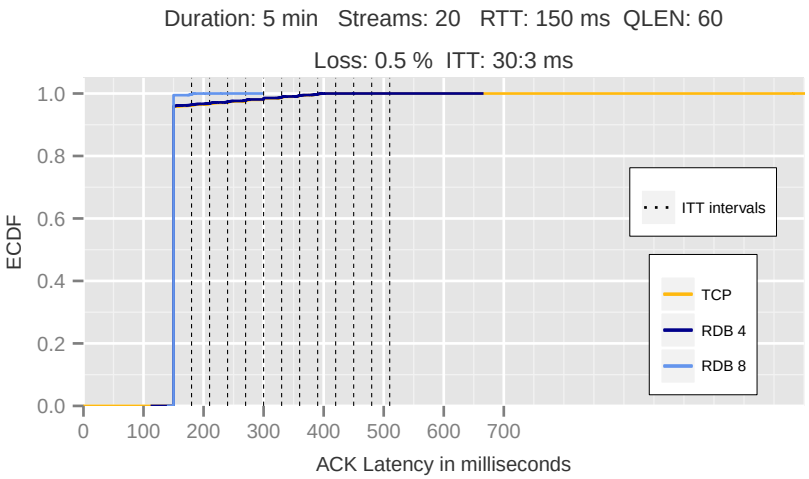
A.1 Latency tests with uniform loss

Table A.1

	Traffic characteristics					Network properties	
	Streams	Payload	ITT (ms)	Type	Max PIF	RTT (ms)	Loss (%)
Figure A.1.1.(a)	20	120	30:3	TCP	NA	150	0.5
Figure A.1.1.(a)	20	120	30:3	RDB	4	150	0.5
Figure A.1.1.(a)	20	120	30:3	RDB	8	150	0.5
Figure A.1.1.(b)	20	120	50:5	TCP	NA	150	0.5
Figure A.1.1.(b)	20	120	50:5	RDB	8	150	0.5
Figure A.1.1.(b)	20	120	50:5	RDB	4	150	0.5
Figure A.1.1.(c)	20	120	75:7	RDB	4	150	0.5
Figure A.1.1.(c)	20	120	75:7	TCP	NA	150	0.5
Figure A.1.1.(c)	20	120	75:7	RDB	8	150	0.5
Figure A.1.1.(d)	20	120	100:10	RDB	8	150	0.5
Figure A.1.1.(d)	20	120	100:10	TCP	NA	150	0.5
Figure A.1.1.(d)	20	120	100:10	RDB	4	150	0.5
Figure A.1.2.(a)	20	120	30:3	TCP	NA	150	2
Figure A.1.2.(a)	20	120	30:3	RDB	4	150	2
Figure A.1.2.(a)	20	120	30:3	RDB	8	150	2
Figure A.1.2.(b)	20	120	50:5	TCP	NA	150	2
Figure A.1.2.(b)	20	120	50:5	RDB	8	150	2
Figure A.1.2.(b)	20	120	50:5	RDB	4	150	2
Figure A.1.2.(c)	20	120	75:7	RDB	4	150	2
Figure A.1.2.(c)	20	120	75:7	TCP	NA	150	2
Figure A.1.2.(c)	20	120	75:7	RDB	8	150	2
Figure A.1.2.(d)	20	120	100:10	RDB	8	150	2
Figure A.1.2.(d)	20	120	100:10	TCP	NA	150	2
Figure A.1.2.(d)	20	120	100:10	RDB	4	150	2
Figure A.1.3.(a)	20	120	30:3	TCP	NA	150	5
Figure A.1.3.(a)	20	120	30:3	RDB	4	150	5
Figure A.1.3.(a)	20	120	30:3	RDB	8	150	5
Figure A.1.3.(b)	20	120	50:5	TCP	NA	150	5

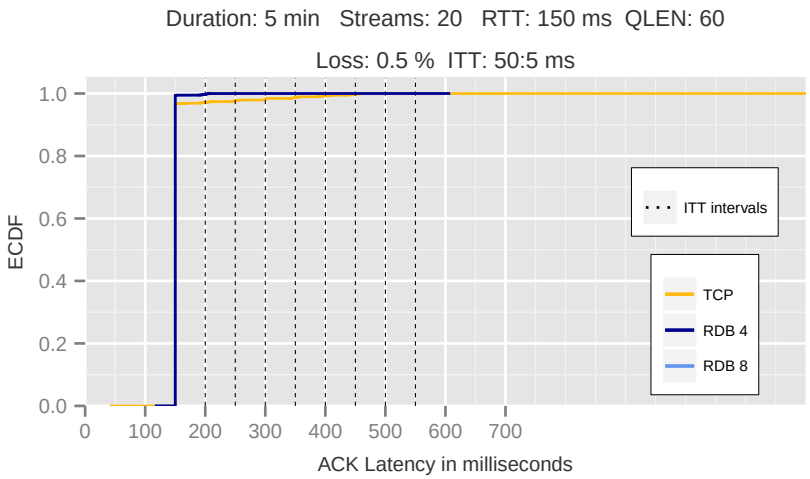
Figure A.1.3.(b)	20	120	50:5	RDB	8	150	5
Figure A.1.3.(b)	20	120	50:5	RDB	4	150	5
Figure A.1.3.(c)	20	120	75:7	RDB	4	150	5
Figure A.1.3.(c)	20	120	75:7	TCP	NA	150	5
Figure A.1.3.(c)	20	120	75:7	RDB	8	150	5
Figure A.1.3.(d)	20	120	100:10	RDB	8	150	5
Figure A.1.3.(d)	20	120	100:10	TCP	NA	150	5
Figure A.1.3.(d)	20	120	100:10	RDB	4	150	5
Figure A.1.4.(a)	20	120	30:3	TCP	NA	150	10
Figure A.1.4.(a)	20	120	30:3	RDB	4	150	10
Figure A.1.4.(a)	20	120	30:3	RDB	8	150	10
Figure A.1.4.(b)	20	120	50:5	TCP	NA	150	10
Figure A.1.4.(b)	20	120	50:5	RDB	8	150	10
Figure A.1.4.(b)	20	120	50:5	RDB	4	150	10
Figure A.1.4.(c)	20	120	75:7	RDB	4	150	10
Figure A.1.4.(c)	20	120	75:7	TCP	NA	150	10
Figure A.1.4.(c)	20	120	75:7	RDB	8	150	10
Figure A.1.4.(d)	20	120	100:10	RDB	8	150	10
Figure A.1.4.(d)	20	120	100:10	TCP	NA	150	10
Figure A.1.4.(d)	20	120	100:10	RDB	4	150	10

Table A.1: Test setup for experiment 1



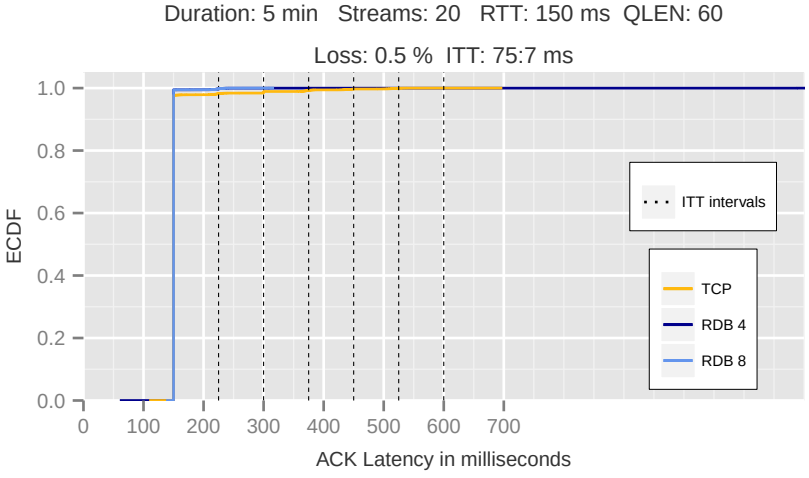
Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	156	150	150	150	150	150	150	150	235	353	394	589	3157
RDB 4	156	150	150	150	150	150	150	150	214	334	393	414	628
RDB 8	151	150	150	150	150	150	150	150	150	150	183	214	288

(a)



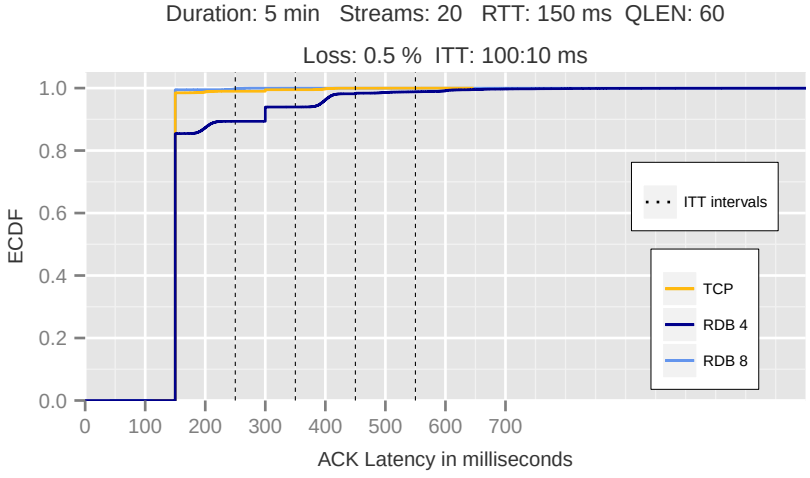
Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	156	150	150	150	150	150	150	150	195	390	458	1239	1510
RDB 4	151	150	150	150	150	150	150	150	150	150	204	304	574
RDB 8	151	150	150	150	150	150	150	150	150	150	204	304	574

(b)



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	155	150	150	150	150	150	150	150	150	366	521	536	657
RDB 4	151	150	150	150	150	150	150	150	150	150	231	1210	1268
RDB 8	151	150	150	150	150	150	150	150	150	150	230	244	305

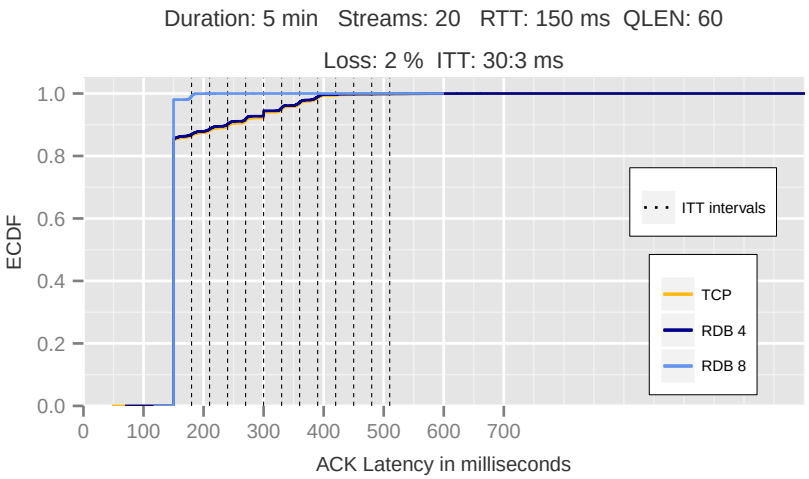
(c)



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	153	150	150	150	150	150	150	150	150	300	411	586	610
RDB 4	178	150	150	150	150	150	300	392	405	591	864	1562	2589
RDB 8	151	150	150	150	150	150	150	150	150	150	260	1377	1511

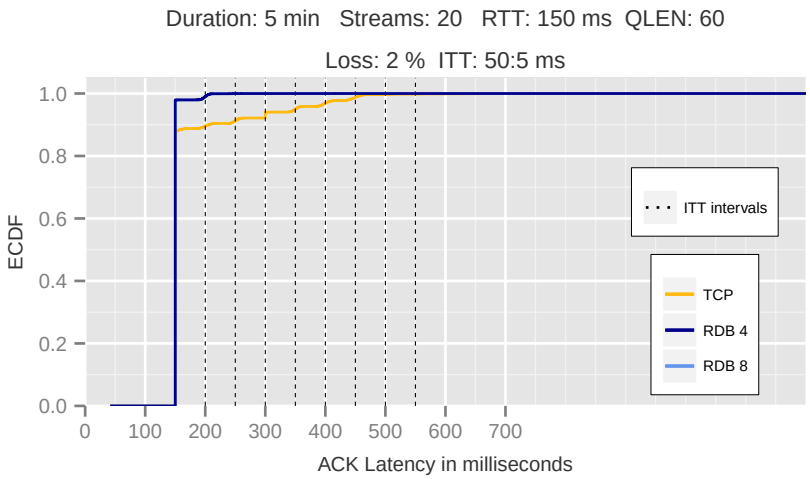
(d)

Figure A.1.1



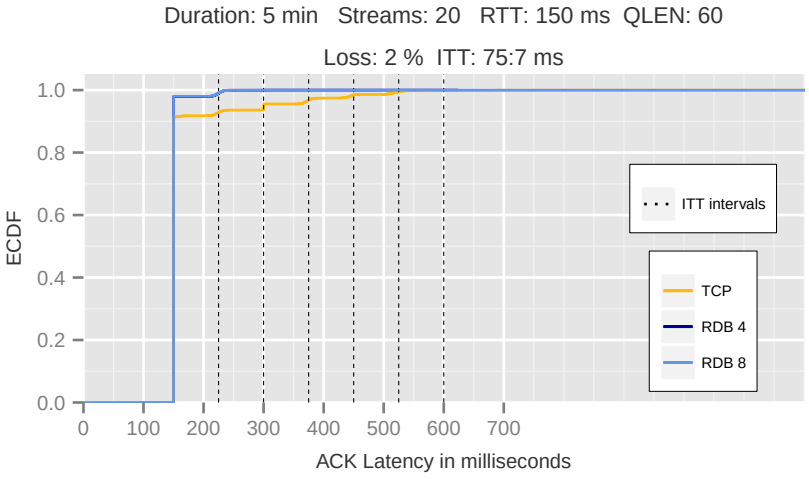
Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	172	150	150	150	150	150	243	330	361	393	572	871	1433
RDB 4	170	150	150	150	150	150	239	328	359	390	532	630	1158
RDB 8	151	150	150	150	150	150	150	150	150	180	186	231	567

(a)



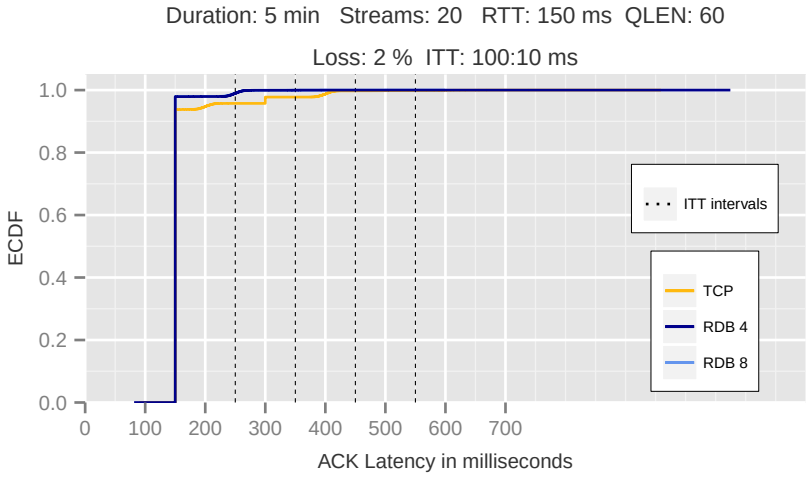
Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	172	150	150	150	150	150	206	350	401	452	610	1261	1505
RDB 4	152	150	150	150	150	150	150	150	150	200	213	1238	1503
RDB 8	152	150	150	150	150	150	150	150	150	200	210	1238	1504

(b)



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	168	150	150	150	150	150	150	300	377	515	615	1150	1802
RDB 4	152	150	150	150	150	150	150	150	150	225	238	304	589
RDB 8	153	150	150	150	150	150	150	150	150	225	239	3215	9667

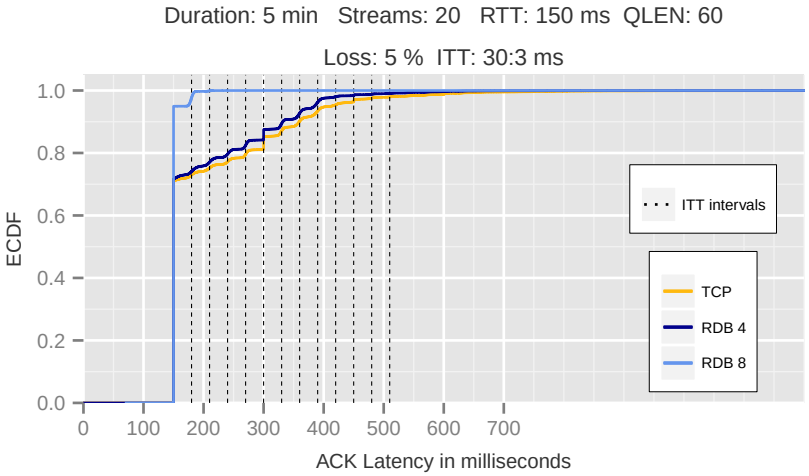
(c)



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	160	150	150	150	150	150	150	204	300	403	592	675	899
RDB 4	153	150	150	150	150	150	150	150	150	251	272	438	1006
RDB 8	153	150	150	150	150	150	150	150	150	250	270	435	566

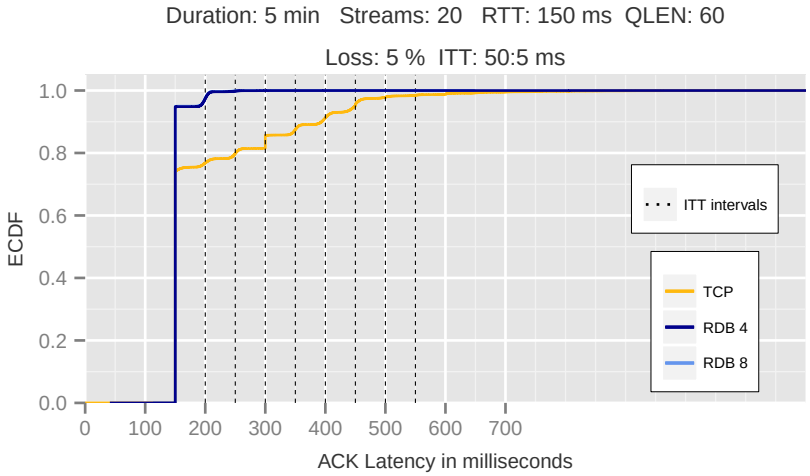
(d)

Figure A.1.2



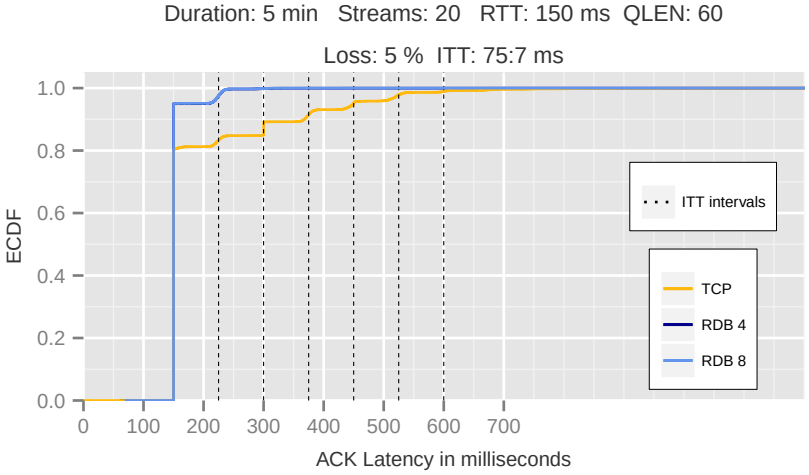
Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	204	150	150	150	150	272	359	413	451	601	954	2013	2510
RDB 4	194	150	150	150	150	242	332	385	394	505	688	1241	3157
RDB 8	152	150	150	150	150	150	150	172	179	183	211	245	1165

(a)



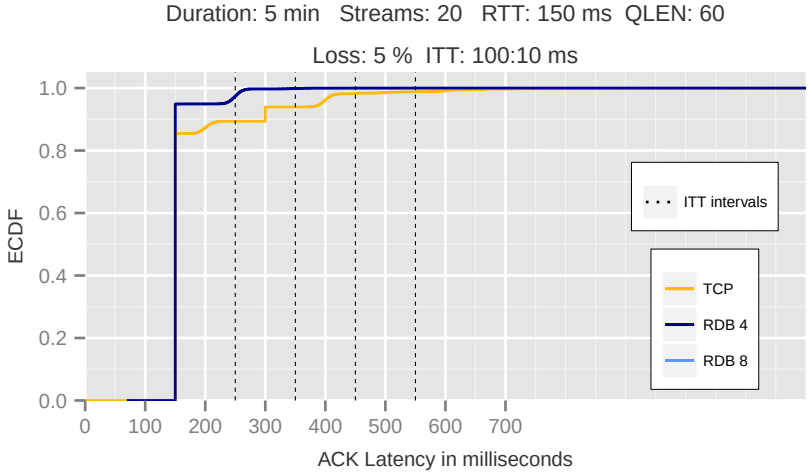
Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	203	150	150	150	150	251	394	449	459	601	900	1966	3106
RDB 4	153	150	150	150	150	150	150	190	199	205	257	1239	1511
RDB 8	153	150	150	150	150	150	150	186	199	205	253	347	612

(b)



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	194	150	150	150	150	150	368	450	518	601	888	1856	2753
RDB 4	155	150	150	150	150	150	150	157	223	232	307	1209	1270
RDB 8	154	150	150	150	150	150	150	150	223	232	305	581	1165

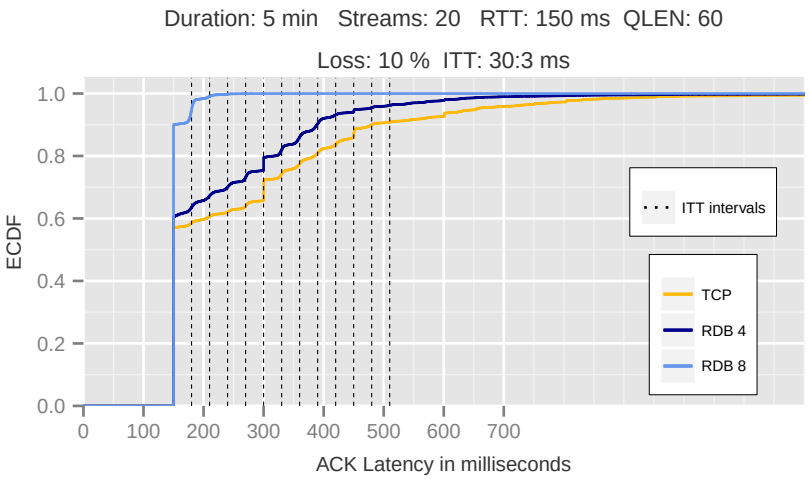
(c)



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	178	150	150	150	150	150	300	392	405	591	864	1562	2589
RDB 4	156	150	150	150	150	150	150	227	248	260	357	463	1158
RDB 8	156	150	150	150	150	150	150	231	248	260	357	487	1177

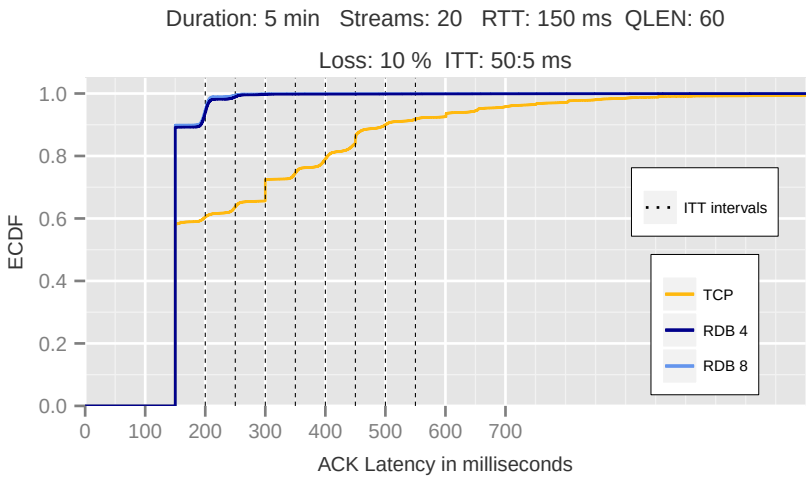
(d)

Figure A.1.3



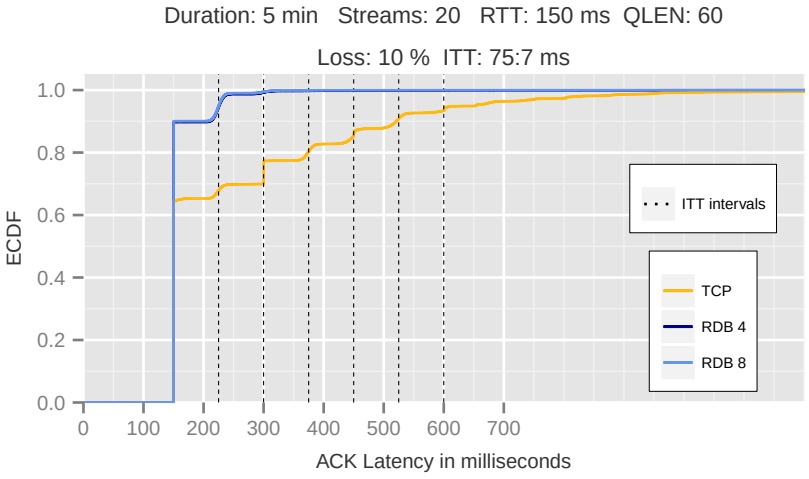
Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	270	150	150	205	300	385	480	654	775	956	2150	2852	4095
RDB 4	227	150	150	150	240	318	389	463	551	717	1511	5495	9650
RDB 8	154	150	150	150	150	150	150	180	183	209	250	813	3154

(a)



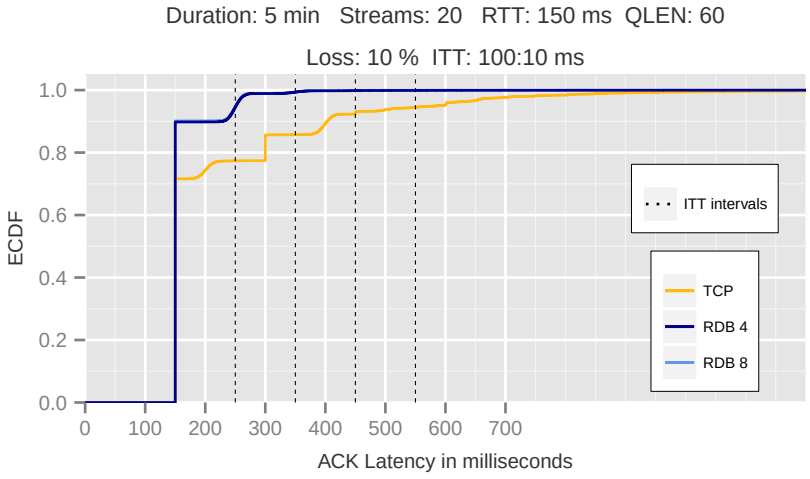
Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	273	150	150	197	300	404	500	655	783	957	2175	3230	4074
RDB 4	158	150	150	150	150	150	193	202	206	251	751	2266	3503
RDB 8	156	150	150	150	150	150	187	200	204	235	296	979	2200

(b)



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	258	150	150	150	300	374	521	653	751	956	1963	2972	3505
RDB 4	161	150	150	150	150	150	211	226	231	294	677	3558	12020
RDB 8	159	150	150	150	150	150	150	225	230	277	373	1241	6106

(c)



Name	Mean	40th	50th	60th	70th	80th	90th	95th	97th	99th	99.9th	99.99th	Max
TCP	223	150	150	150	150	300	403	588	655	874	1696	2541	3156
RDB 4	163	150	150	150	150	150	229	251	258	333	590	2198	6105
RDB 8	162	150	150	150	150	150	150	251	257	324	455	639	1380

(d)

Figure A.1.4

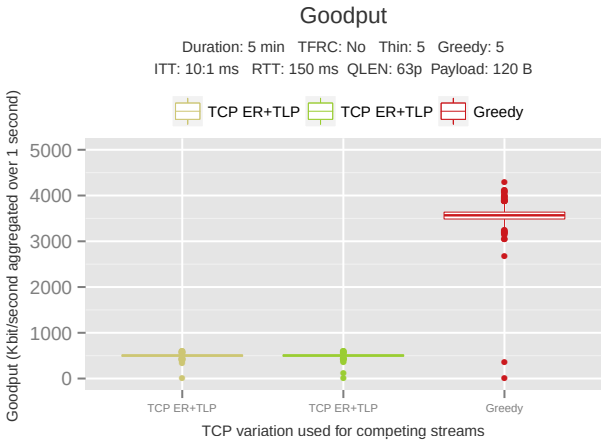
A.2 Latency tests with greedy cross traffic

Table A.2

	Type	Streams	Cong	TFRC	RTT	ITT	Payload
Figure A.2.1	TCP ER+TLP	5	Rdb	No	150	10:1	120
Figure A.2.2	RDB DPIF10 ER+TLP	5	Rdb	No	150	10:1	120
Figure A.2.3	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	10:1	120
Figure A.2.4	RDB DPIF20 ER+TLP	5	Rdb	No	150	10:1	120
Figure A.2.5	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	10:1	120
Figure A.2.6	TCP ER+TLP	5	Rdb	No	150	30:3	120
Figure A.2.7	RDB DPIF10 ER+TLP	5	Rdb	No	150	30:3	120
Figure A.2.8	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	30:3	120
Figure A.2.9	RDB DPIF20 ER+TLP	5	Rdb	No	150	30:3	120
Figure A.2.10	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	30:3	120
Figure A.2.11	TCP ER+TLP	5	Rdb	No	150	50:5	120
Figure A.2.12	RDB DPIF10 ER+TLP	5	Rdb	No	150	50:5	120
Figure A.2.13	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	50:5	120
Figure A.2.14	RDB DPIF20 ER+TLP	5	Rdb	No	150	50:5	120
Figure A.2.15	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	50:5	120
Figure A.2.16	TCP ER+TLP	5	Rdb	No	150	75:7	120
Figure A.2.17	RDB DPIF10 ER+TLP	5	Rdb	No	150	75:7	120
Figure A.2.18	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	75:7	120
Figure A.2.19	RDB DPIF20 ER+TLP	5	Rdb	No	150	75:7	120
Figure A.2.20	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	75:7	120
Figure A.2.21	TCP ER+TLP	10	Rdb	No	150	10:1	120
Figure A.2.22	RDB DPIF10 ER+TLP	10	Rdb	No	150	10:1	120
Figure A.2.23	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	10:1	120
Figure A.2.24	RDB DPIF20 ER+TLP	10	Rdb	No	150	10:1	120
Figure A.2.25	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	10:1	120
Figure A.2.26	TCP ER+TLP	10	Rdb	No	150	30:3	120
Figure A.2.27	RDB DPIF10 ER+TLP	10	Rdb	No	150	30:3	120
Figure A.2.28	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	30:3	120
Figure A.2.29	RDB DPIF20 ER+TLP	10	Rdb	No	150	30:3	120
Figure A.2.30	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	30:3	120
Figure A.2.31	TCP ER+TLP	10	Rdb	No	150	50:5	120
Figure A.2.32	RDB DPIF10 ER+TLP	10	Rdb	No	150	50:5	120
Figure A.2.33	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	50:5	120
Figure A.2.34	RDB DPIF20 ER+TLP	10	Rdb	No	150	50:5	120
Figure A.2.35	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	50:5	120
Figure A.2.36	TCP ER+TLP	10	Rdb	No	150	75:7	120
Figure A.2.37	RDB DPIF10 ER+TLP	10	Rdb	No	150	75:7	120
Figure A.2.38	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	75:7	120
Figure A.2.39	RDB DPIF20 ER+TLP	10	Rdb	No	150	75:7	120
Figure A.2.40	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	75:7	120
Figure A.2.41	TCP ER+TLP	16	Rdb	No	150	10:1	120
Figure A.2.42	RDB DPIF10 ER+TLP	16	Rdb	No	150	10:1	120
Figure A.2.43	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	10:1	120
Figure A.2.44	RDB DPIF20 ER+TLP	16	Rdb	No	150	10:1	120
Figure A.2.45	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	10:1	120
Figure A.2.46	TCP ER+TLP	16	Rdb	No	150	30:3	120

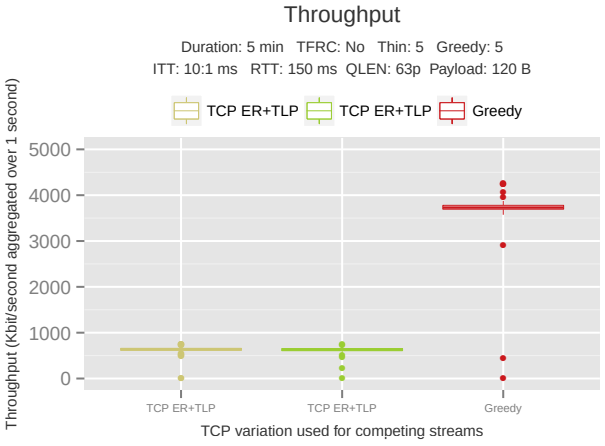
Figure A.2.47	RDB DPIF10 ER+TLP	16	Rdb	No	150	30:3	120
Figure A.2.48	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	30:3	120
Figure A.2.49	RDB DPIF20 ER+TLP	16	Rdb	No	150	30:3	120
Figure A.2.50	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	30:3	120
Figure A.2.51	TCP ER+TLP	16	Rdb	No	150	50:5	120
Figure A.2.52	RDB DPIF10 ER+TLP	16	Rdb	No	150	50:5	120
Figure A.2.53	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	50:5	120
Figure A.2.54	RDB DPIF20 ER+TLP	16	Rdb	No	150	50:5	120
Figure A.2.55	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	50:5	120
Figure A.2.56	TCP ER+TLP	16	Rdb	No	150	75:7	120
Figure A.2.57	RDB DPIF10 ER+TLP	16	Rdb	No	150	75:7	120
Figure A.2.58	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	75:7	120
Figure A.2.59	RDB DPIF20 ER+TLP	16	Rdb	No	150	75:7	120
Figure A.2.60	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	75:7	120

Table A.2: Test setup for experiment 2



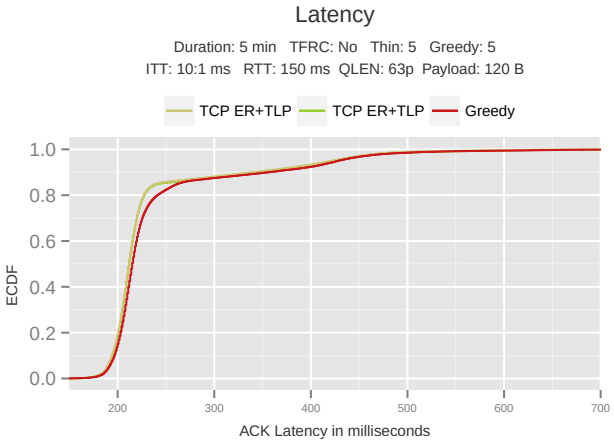
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	17 / 19 / 21	223/246/275	13/1/9/13/17/28
TCP ER+TLP	rdbsender	5	18 / 20 / 22	229/256/285	13/1/9/12/16/28
Greedy	zsender	5	15 / 16 / 16	1446/1447/1447	14/3/10/13/16/36

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
78.1 K	1117	1119	0	19.1 M	18.9 M	244 K	244 K	1.28 / 1.43	11055
73.9 K	1134	1136	0	18.8 M	18.5 M	248 K	248 K	1.32 / 1.53	10672
94.6 K	1577	1578	0	137 M	135 M	2.3 M	2.3 M	1.66 / 1.67	12820

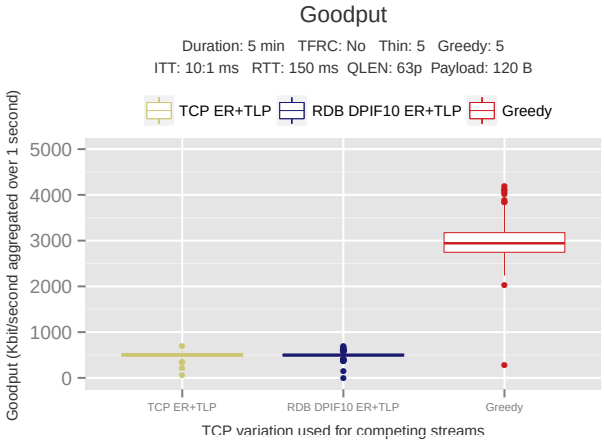
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
236	212	216	220	227	340	395	422	434	449	466	506	674	1052	1319
237	212	215	220	228	349	401	426	438	453	474	529	712	1025	1152
242	215	220	226	241	356	408	428	439	454	477	530	727	1088	1152

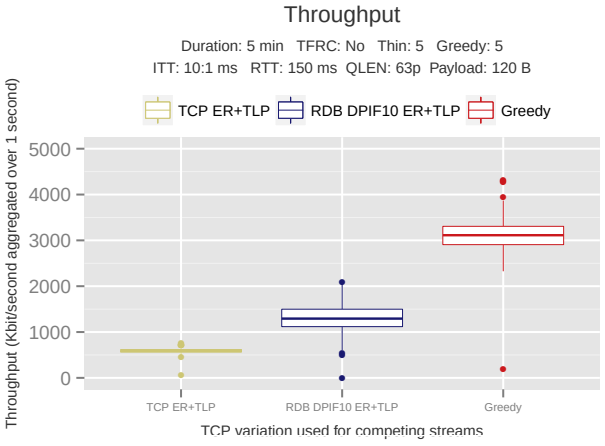
(c)

Figure A.2.1



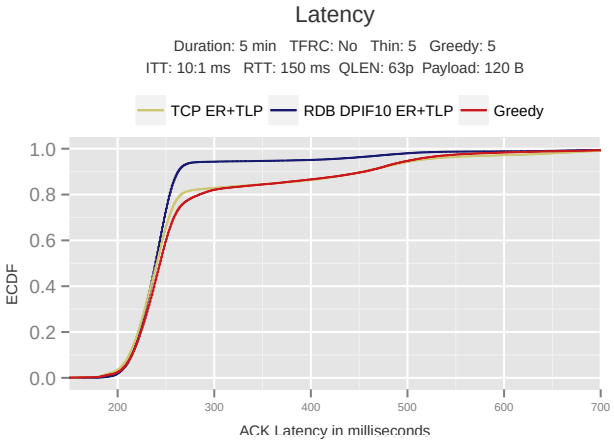
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	24 / 28 / 32	315/367/407	10/1/7/10/13/30
RDB DPIF10 ER+TLP	rdbsender	5	28 / 33 / 41	1001/1036/1057	10/1/6/9/12/28
Greedy	zsender	5	16 / 19 / 22	1446/1447/1447	15/1/10/13/18/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.6 K	1340	1340	0	19.2 M	18.8 M	334 K	334 K	1.74 / 2.55	8708
45.2 K	1262	428	31.7 K	46.8 M	18.6 M	28.2 M	307 K	2.76 / 2.79	2385
79.7 K	1764	1764	0	115 M	113 M	2.6 M	2.6 M	2.21 / 2.21	13174

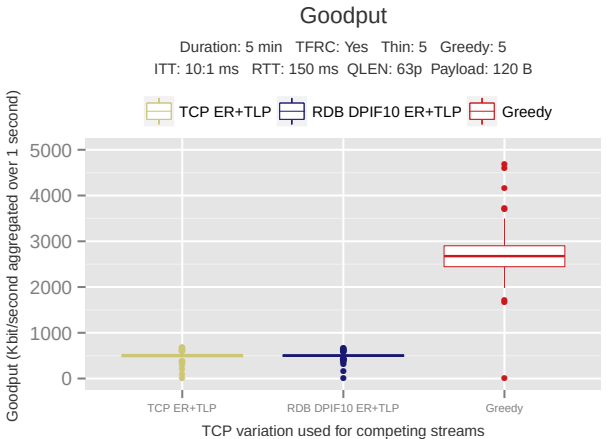
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
279	241	246	253	266	454	486	511	534	587	651	696	1228	1678	1782
253	239	244	248	254	263	270	391	441	470	500	638	1030	1481	1775
280	244	250	258	285	455	483	504	517	537	576	665	864	1113	1522

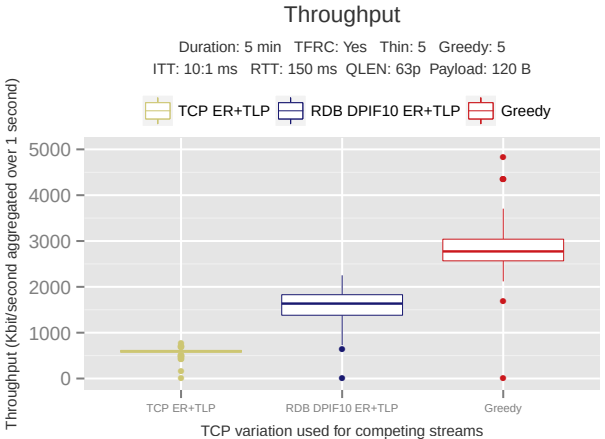
(c)

Figure A.2.2



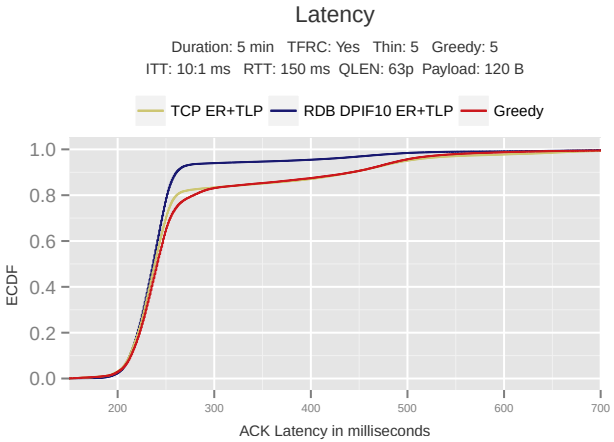
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	26 / 28 / 31	333/365/398	10/1/7/10/13/28
RDB DPIF10 ER+TLP	rdbsender	5	24 / 25 / 26	888/959/1023	12/1/8/11/14/86
Greedy	zsender	5	19 / 21 / 23	1446/1447/1448	12/1/9/11/15/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.8 K	1290	1292	0	19.2 M	18.8 M	385 K	385 K	2.00 / 2.44	8764
59.5 K	1504	449	42.9 K	56.9 M	18.6 M	38.4 M	190 K	2.51 / 2.53	3617
72.2 K	1614	1616	0	104 M	102 M	2.3 M	2.3 M	2.23 / 2.24	11405

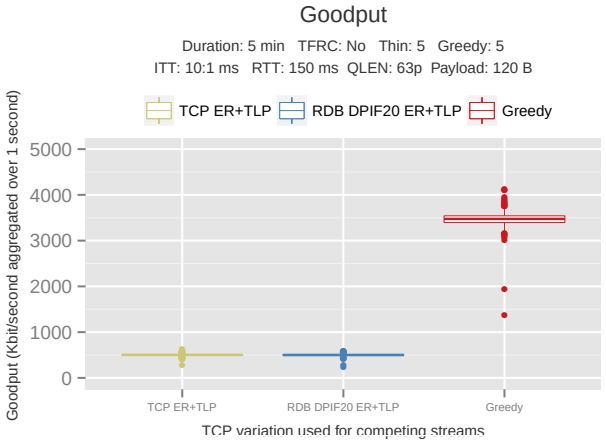
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
274	238	244	250	261	443	475	499	516	548	616	674	1018	1725	2420
250	237	241	246	251	261	272	374	422	453	484	557	934	1592	1871
278	241	247	255	279	442	473	492	504	522	552	623	931	10073	13590

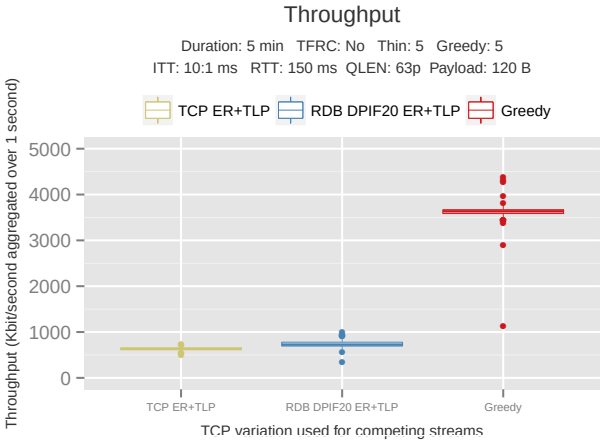
(c)

Figure A.2.3



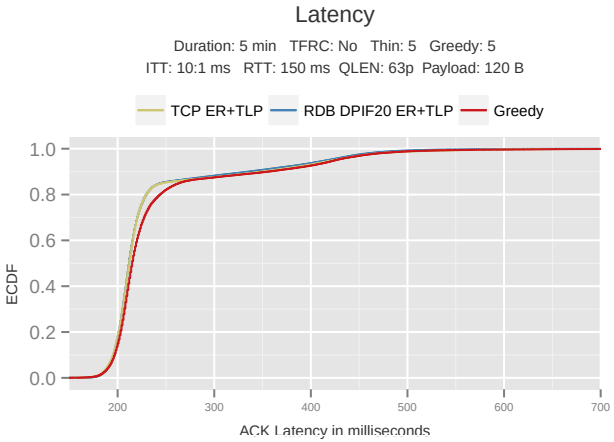
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	19 / 19 / 20	247/250/253	13/1/9/12/16/29
RDB DPIF20 ER+TLP	rdbsender	5	18 / 18 / 18	274/279/287	13/1/10/13/16/28
Greedy	zsender	5	15 / 16 / 17	1446/1447/1447	14/4/10/13/16/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
76.4 K	1119	1119	0	19.1 M	18.9 M	215 K	215 K	1.13 / 1.46	11078
80.8 K	1177	1107	4.8 K	22.6 M	18.6 M	3.9 M	208 K	1.27 / 1.46	11751
92.0 K	1555	1555	0	133 M	131 M	2.2 M	2.2 M	1.68 / 1.69	12414

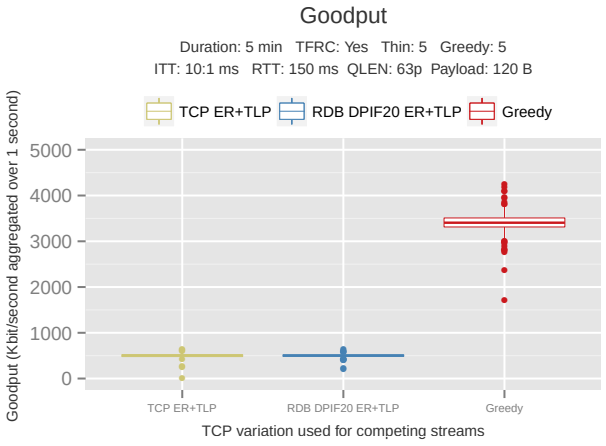
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
237	212	216	221	230	350	399	424	436	451	470	517	688	902	999
235	212	215	220	230	336	390	417	430	443	460	488	614	847	999
241	215	220	227	244	356	405	426	437	451	471	509	719	929	1012

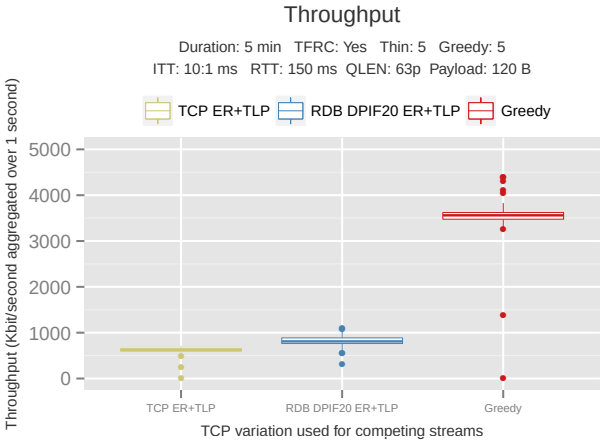
(c)

Figure A.2.4



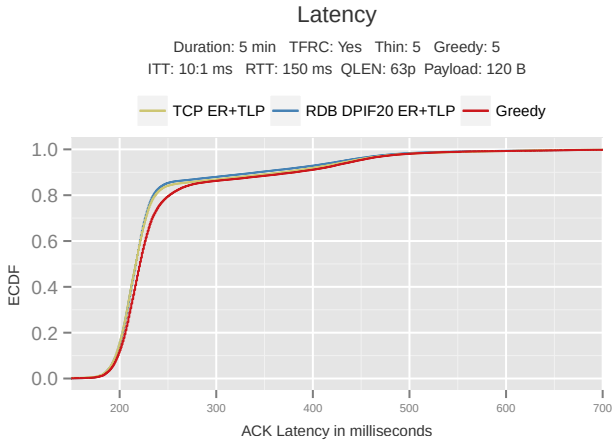
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	19 / 21 / 24	247/272/312	12/1/9/12/15/27
RDB DPIF20 ER+TLP	rdbsender	5	21 / 22 / 23	373/392/418	14/1/8/12/18/39
Greedy	zsender	5	16 / 16 / 17	1447/1447/1447	14/1/10/13/16/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
70.5 K	1191	1193	0	19.1 M	18.8 M	249 K	249 K	1.30 / 1.69	10415
68.0 K	1122	946	10.0 K	26.6 M	18.6 M	8.0 M	220 K	1.62 / 1.65	9304
90.4 K	1686	1686	0	131 M	128 M	2.4 M	2.4 M	1.86 / 1.86	12882

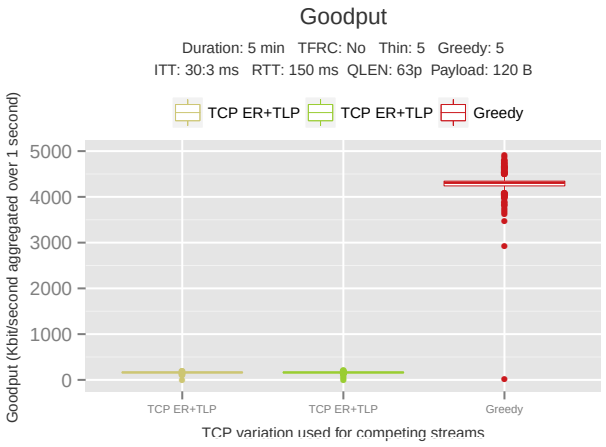
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
244	217	221	227	237	368	416	440	453	469	496	570	754	1254	1429
241	217	221	227	235	343	402	432	446	463	487	546	969	1213	1239
250	221	227	234	252	381	424	446	457	473	497	555	745	3869	4091

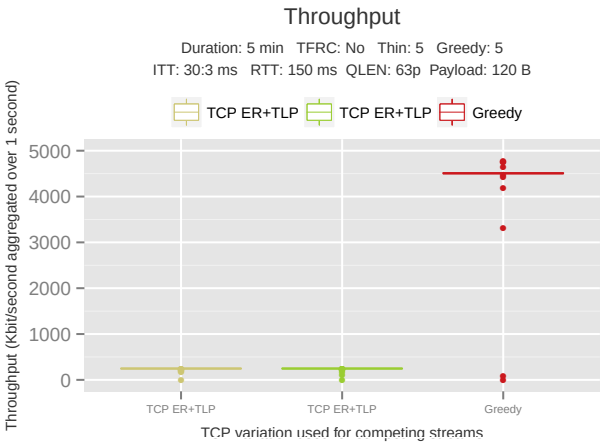
(c)

Figure A.2.5



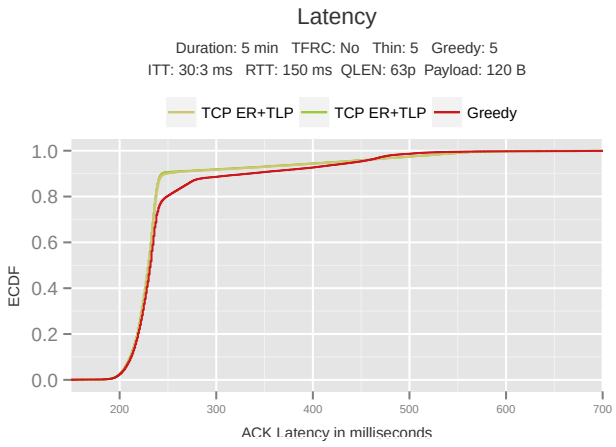
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 32	126/127/130	11/1/8/12/13/14
TCP ER+TLP	rdbsender	5	31 / 31 / 31	126/127/128	11/3/9/12/13/13
Greedy	zsender	5	12 / 13 / 14	1446/1447/1448	18/6/14/17/21/50

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.4 K	540	542	0	6.2 M	6.1 M	65.5 K	65.5 K	1.06 / 1.12	4205
47.8 K	532	534	0	6.1 M	6.0 M	64.3 K	64.3 K	1.06 / 1.11	4062
113 K	1175	1177	0	164 M	162 M	1.7 M	1.7 M	1.04 / 1.04	13889

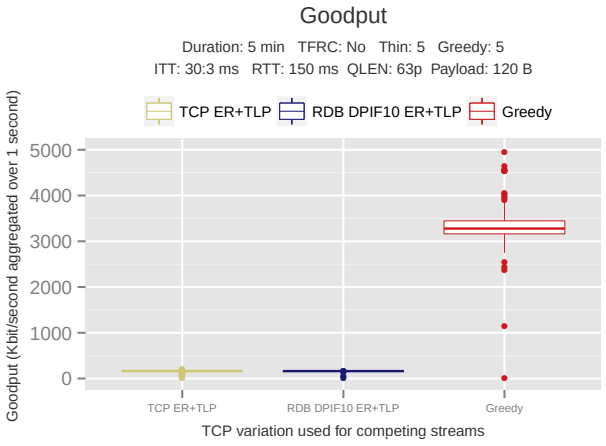
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	230	233	235	238	249	353	425	458	489	520	549	616	841	1152
245	230	233	235	237	244	347	419	454	487	519	551	664	890	1152
253	232	235	238	249	333	406	444	459	469	481	510	691	900	1152

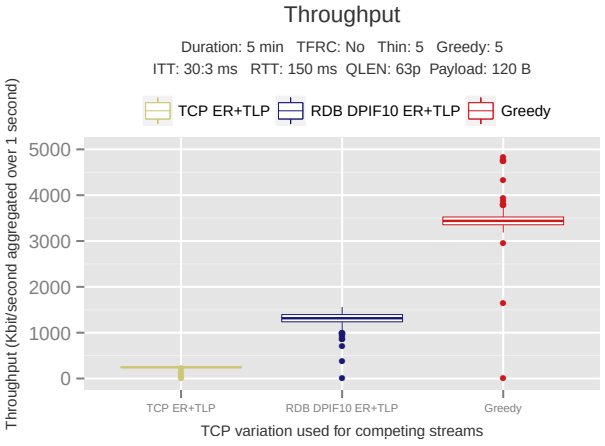
(c)

Figure A.2.6



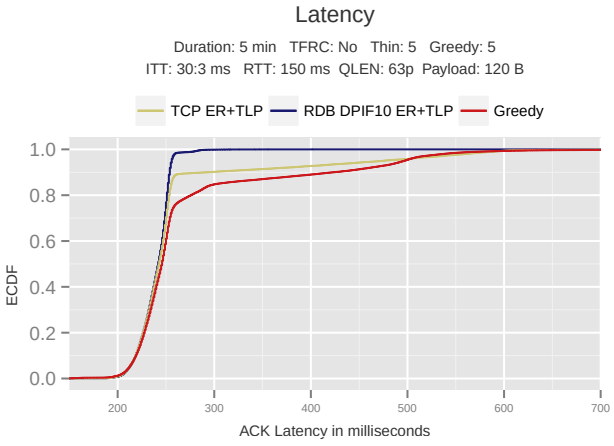
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 32 / 32	128/130/130	10/2/8/11/13/14
RDB DPIF10 ER+TLP	rdbsender	5	32 / 33 / 33	1015/1017/1018	10/2/8/11/13/14
Greedy	zsender	5	15 / 17 / 18	1447/1447/1447	14/4/11/14/16/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
47.5 K	611	613	0	6.2 M	6.1 M	74.7 K	74.7 K	1.21 / 1.28	4615
45.2 K	568	2	45.2 K	46.0 M	6.0 M	40.0 M	0	1.31 / 1.26	
87.8 K	1458	1460	0	127 M	125 M	2.1 M	2.1 M	1.66 / 1.66	12759

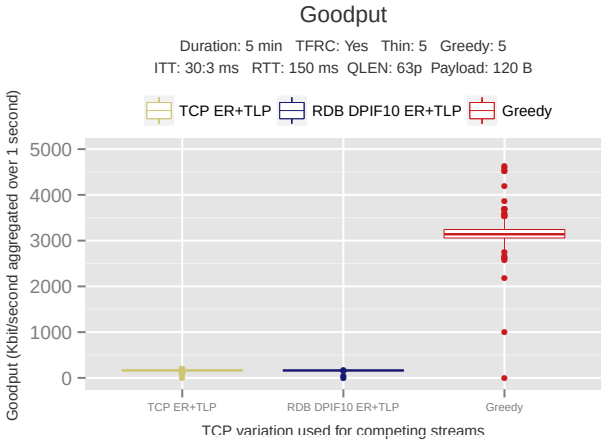
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
262	243	247	250	253	293	410	475	504	534	561	589	733	1154	1777
240	242	246	249	251	254	255	256	257	258	259	278	308	385	1153
274	246	250	254	276	423	477	497	504	516	538	575	785	1015	1151

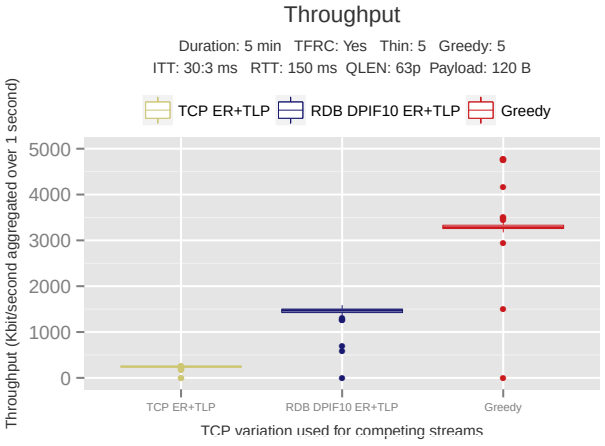
(c)

Figure A.2.7



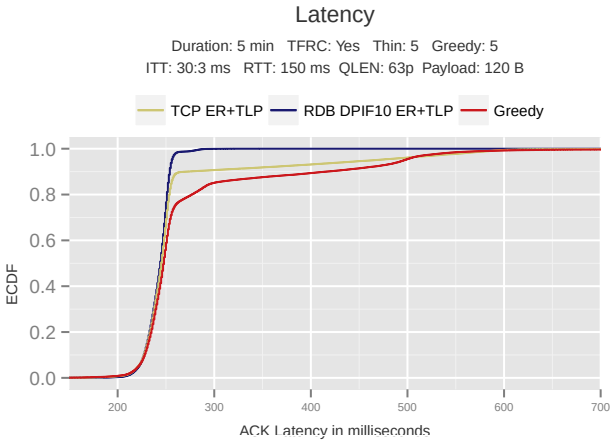
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 32 / 32	128/131/134	11/1/8/12/13/14
RDB DPIF10 ER+TLP	rdbsender	5	30 / 30 / 30	1041/1042/1043	13/4/11/13/13/49
Greedy	zsender	5	17 / 18 / 18	1447/1447/1447	14/2/11/13/16/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
47.3 K	549	551	0	6.2 M	6.1 M	67.8 K	67.8 K	1.10 / 1.16	4385
49.5 K	589	2	49.5 K	51.5 M	6.0 M	45.5 M	0	1.23 / 1.19	
83.8 K	1306	1307	0	121 M	119 M	1.9 M	1.9 M	1.55 / 1.56	11823

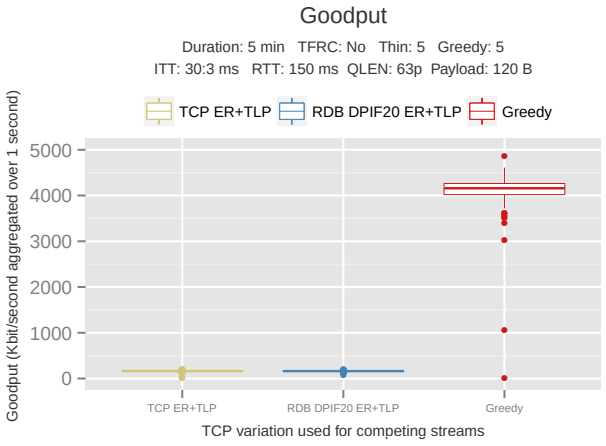
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
264	245	248	251	254	272	395	466	497	527	558	586	753	1163	1302
243	244	247	249	251	254	255	256	257	258	260	276	294	336	1152
277	248	251	255	277	416	478	497	504	515	537	580	800	1028	1152

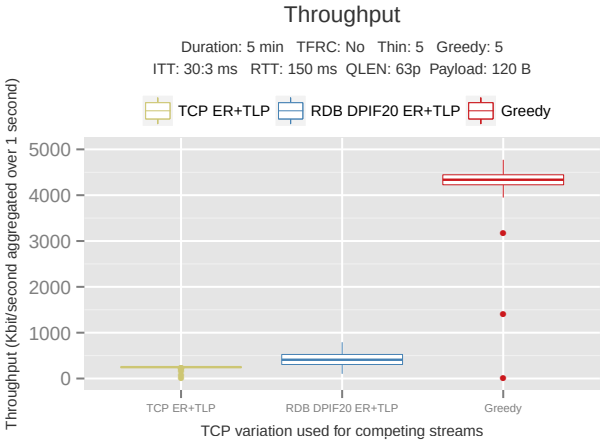
(c)

Figure A.2.8



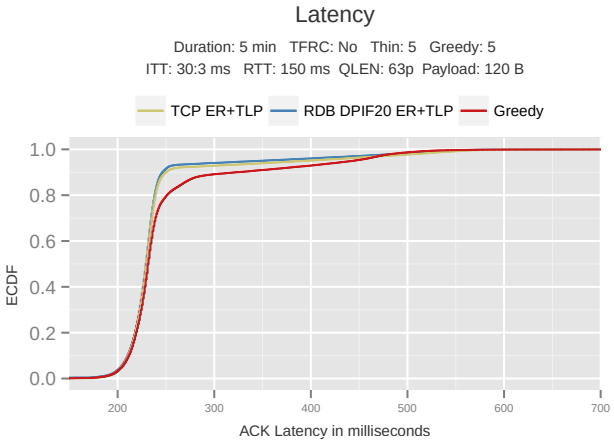
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	30 / 31 / 31	124/126/128	11/1/9/12/13/14
RDB DPIF20 ER+TLP	rdbsender	5	30 / 31 / 31	259/268/275	11/2/9/12/13/13
Greedy	zsender	5	13 / 14 / 14	1446/1447/1447	17/3/14/17/20/44

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.7 K	489	491	0	6.1 M	6.1 M	59.4 K	59.4 K	0.97 / 1.00	3668
48.1 K	473	387	9.0 K	12.9 M	6.0 M	6.9 M	47.0 K	1.00 / 0.98	3053
109 K	1182	1184	0	158 M	156 M	1.7 M	1.7 M	1.08 / 1.08	12729

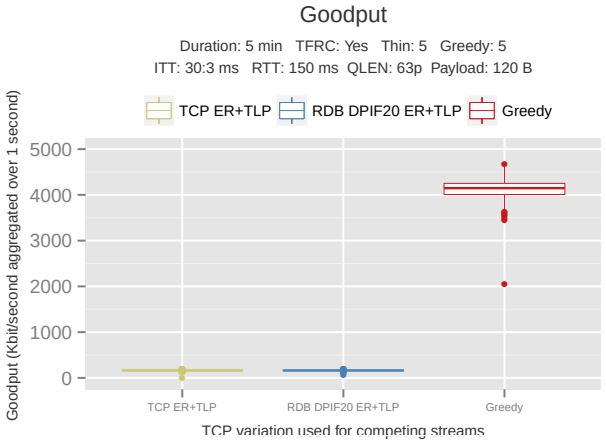
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
244	230	233	236	239	250	307	395	435	472	509	544	605	712	1152
240	229	232	235	238	246	257	347	398	444	490	532	588	714	869
252	232	235	239	251	323	402	444	459	470	484	510	612	959	1152

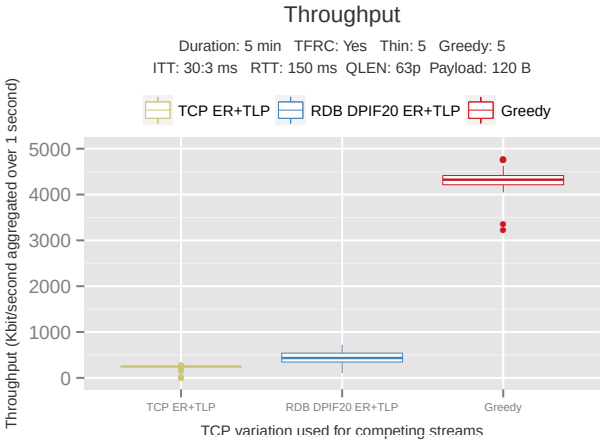
(c)

Figure A.2.9



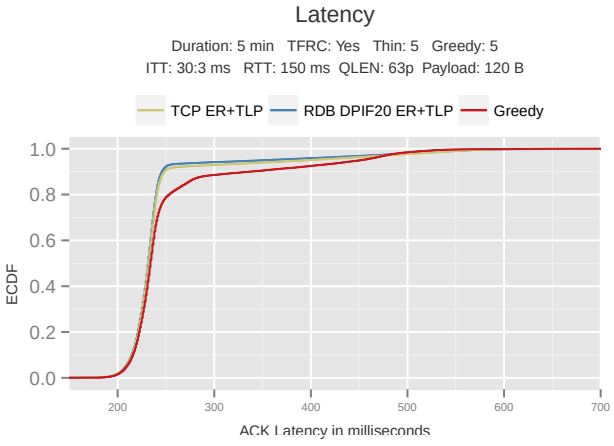
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 31	126/126/127	11/2/9/12/13/14
RDB DPIF20 ER+TLP	rdbsender	5	32 / 32 / 33	280/295/305	13/2/9/12/13/32
Greedy	zsender	5	13 / 14 / 14	1447/1447/1448	17/6/14/17/20/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.7 K	454	456	0	6.2 M	6.1 M	55.0 K	55.0 K	0.89 / 0.93	3640
45.8 K	430	351	9.6 K	13.5 M	6.0 M	7.5 M	42.1 K	0.91 / 0.94	2747
109 K	1182	1182	0	157 M	156 M	1.7 M	1.7 M	1.09 / 1.09	13262

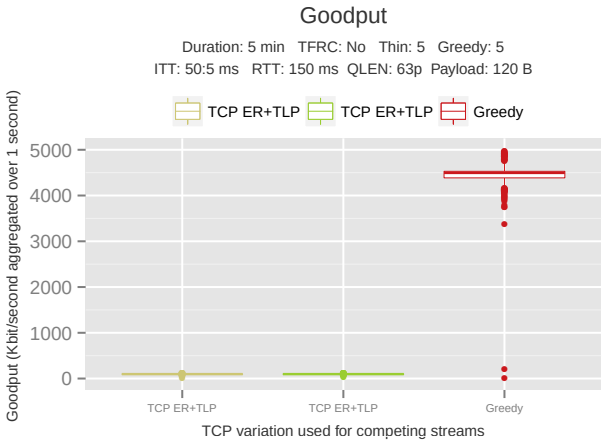
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	232	235	237	241	249	304	397	438	475	513	551	652	905	1244
243	232	234	237	240	246	254	353	407	458	504	548	656	924	1153
256	234	237	241	253	337	412	452	465	477	491	516	624	748	999

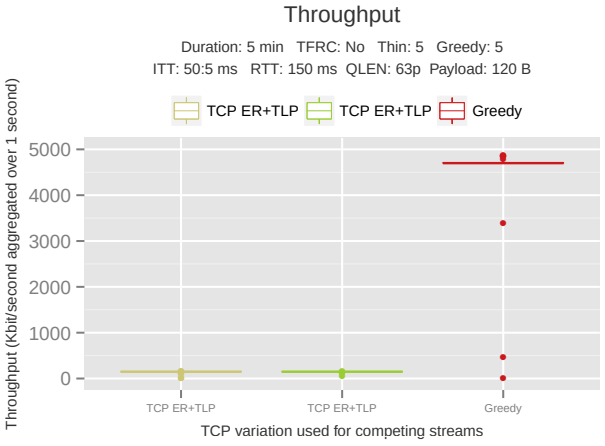
(c)

Figure A.2.10



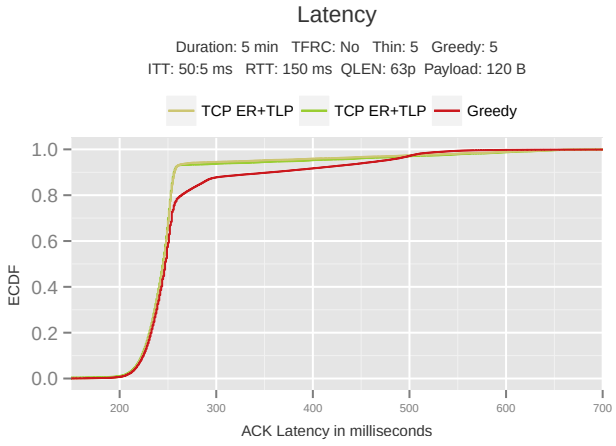
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	121/121/122	9/1/9/10/10/10
TCP ER+TLP	rdbsender	5	50 / 50 / 50	121/122/122	9/1/8/10/10/10
Greedy	zsender	5	12 / 13 / 14	1446/1447/1447	20/4/16/19/23/47

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.2 K	256	258	0	3.7 M	3.6 M	30.7 K	30.7 K	0.84 / 0.85	1632
29.9 K	290	290	0	3.6 M	3.6 M	34.8 K	34.8 K	0.96 / 0.97	1852
118 K	1111	1113	0	171 M	169 M	1.6 M	1.6 M	0.94 / 0.94	14696

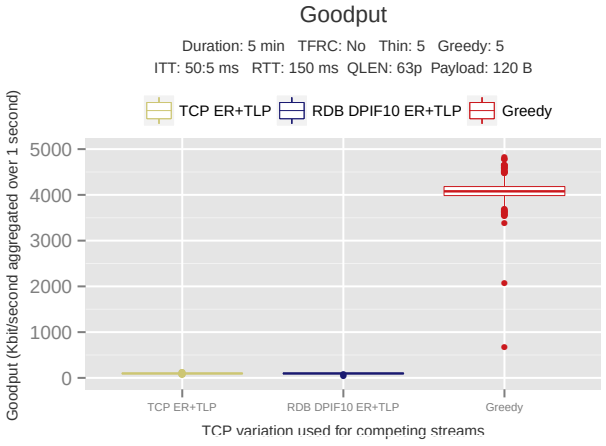
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
255	245	248	251	253	257	261	336	403	477	542	604	698	1000	1153
257	245	248	250	253	256	261	389	446	501	558	613	684	961	1058
268	247	251	254	264	357	431	471	487	499	509	535	695	791	1152

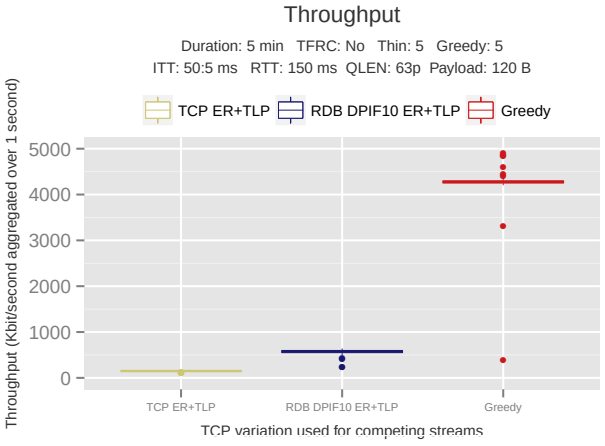
(c)

Figure A.2.11



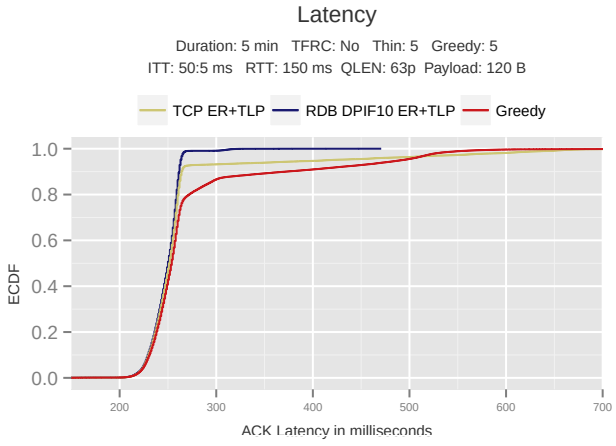
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	122/122/123	9/1/8/10/10/10
RDB DPIF10 ER+TLP	rdbsender	5	50 / 50 / 50	659/660/660	9/2/8/10/10/10
Greedy	zsender	5	12 / 14 / 14	1446/1447/1447	19/6/15/18/21/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.1 K	324	324	0	3.7 M	3.6 M	38.9 K	38.9 K	1.06 / 1.08	1971
29.6 K	278	0	29.6 K	19.5 M	3.6 M	16.0 M	0	0.98 / 0.94	
108 K	1127	1127	0	156 M	154 M	1.6 M	1.6 M	1.05 / 1.05	13591

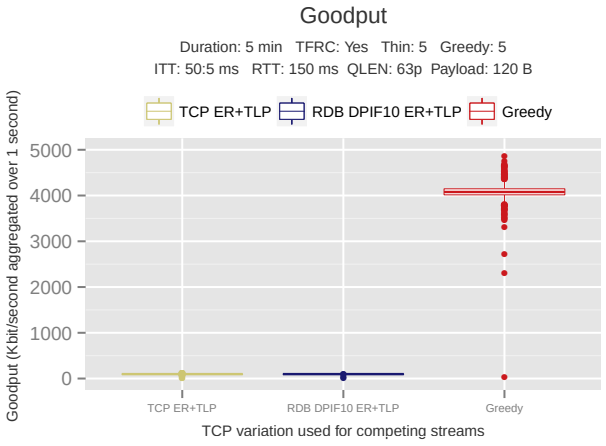
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
266	252	255	258	261	264	285	416	473	531	590	642	743	1002	1141
248	250	254	257	259	262	263	263	264	265	266	270	321	381	447
277	254	258	261	272	372	453	492	506	516	528	554	737	830	999

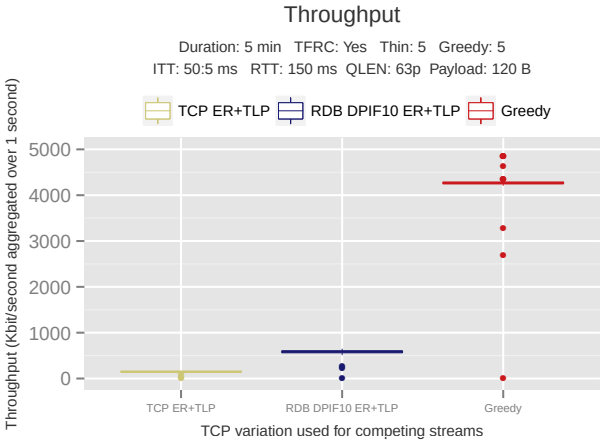
(c)

Figure A.2.12



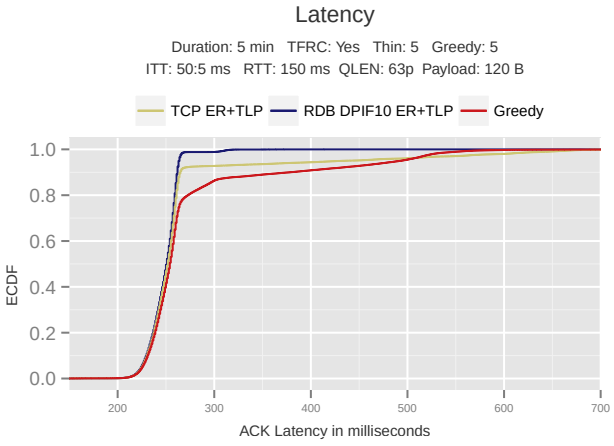
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	122/122/123	9/1/8/10/10/11
RDB DPIF10 ER+TLP	rdbsender	5	50 / 50 / 50	663/664/666	11/4/10/10/12/54
Greedy	zsender	5	13 / 14 / 14	1447/1447/1447	18/7/15/18/21/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.1 K	339	341	0	3.7 M	3.6 M	40.8 K	40.8 K	1.11 / 1.13	2064
29.8 K	346	2	29.8 K	19.8 M	3.6 M	16.3 M	0	1.22 / 1.16	
108 K	1112	1113	0	156 M	154 M	1.6 M	1.6 M	1.03 / 1.03	13800

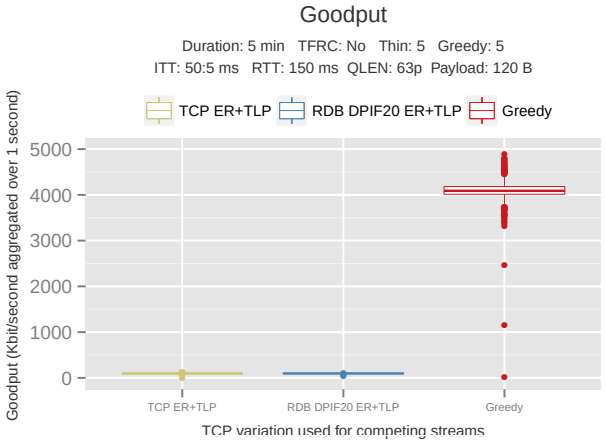
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
267	252	255	258	261	265	314	433	495	546	597	645	716	946	1153
249	251	254	257	259	262	263	264	264	265	266	304	320	414	1152
277	254	258	261	273	378	454	493	506	516	526	551	680	955	1151

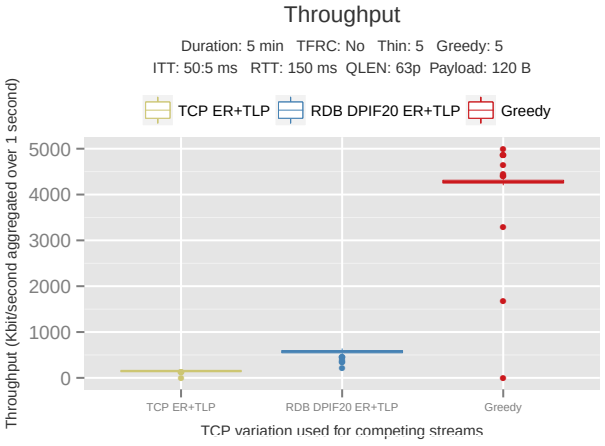
(c)

Figure A.2.13



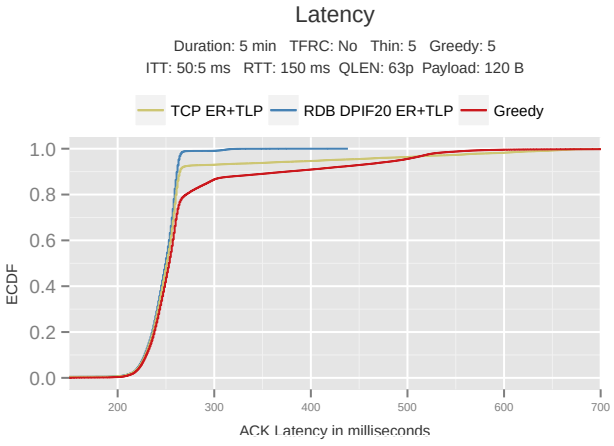
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	122/122/122	9/1/8/10/10/10
RDB DPIF20 ER+TLP	rdbsender	5	50 / 50 / 51	655/657/658	9/2/8/10/10/10
Greedy	zsender	5	13 / 14 / 15	1447/1447/1447	19/5/14/18/22/44

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.1 K	327	328	0	3.7 M	3.6 M	39.1 K	39.1 K	1.06 / 1.08	2037
29.5 K	290	0	29.5 K	19.4 M	3.6 M	15.8 M	0	1.05 / 0.98	1
108 K	1101	1103	0	156 M	155 M	1.6 M	1.6 M	1.02 / 1.02	13712

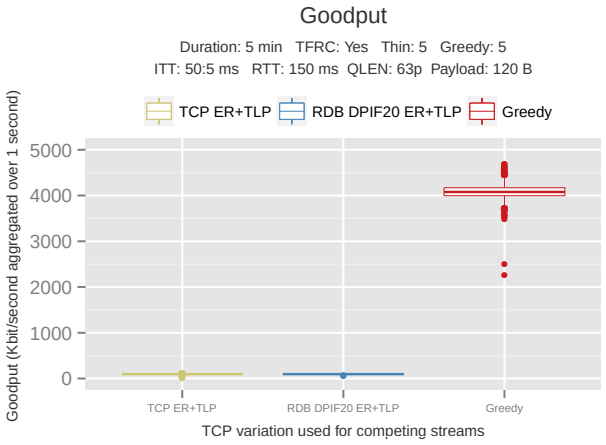
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
265	251	255	258	261	265	299	417	473	528	585	637	709	906	955
248	250	253	257	259	262	263	263	264	265	266	276	321	370	417
277	253	257	261	271	375	453	492	505	516	528	559	754	828	1150

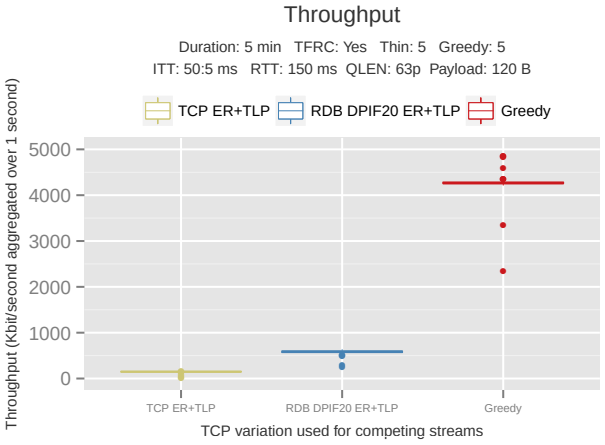
(c)

Figure A.2.14



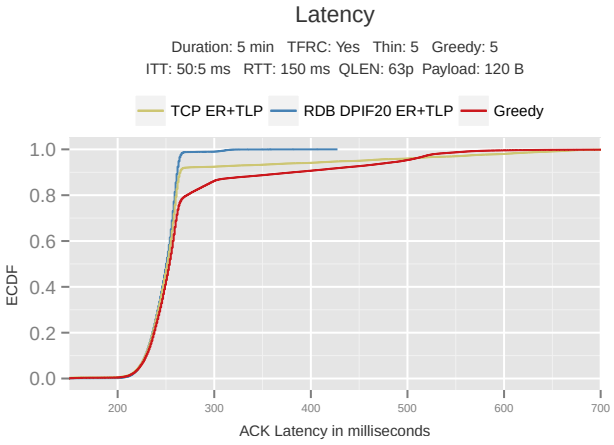
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	122/122/122	9/1/8/10/10/11
RDB DPIF20 ER+TLP	rdbsender	5	49 / 50 / 50	662/663/665	12/5/10/10/12/28
Greedy	zsender	5	13 / 14 / 14	1447/1447/1447	18/7/15/18/21/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.1 K	352	354	0	3.7 M	3.6 M	42.5 K	42.5 K	1.15 / 1.17	2153
29.9 K	315	0	29.9 K	19.9 M	3.6 M	16.3 M	0	1.11 / 1.05	
108 K	1158	1158	0	156 M	154 M	1.7 M	1.7 M	1.08 / 1.08	14049

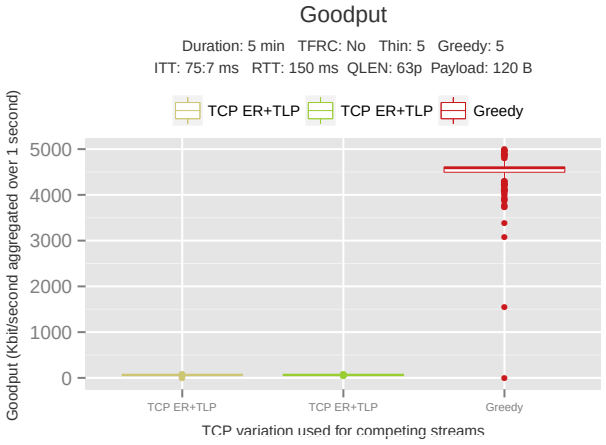
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
267	251	255	258	261	265	329	448	500	551	599	645	716	1016	1285
248	250	254	257	259	262	263	264	264	265	266	300	320	368	407
277	254	257	261	272	382	458	496	508	517	529	560	739	890	999

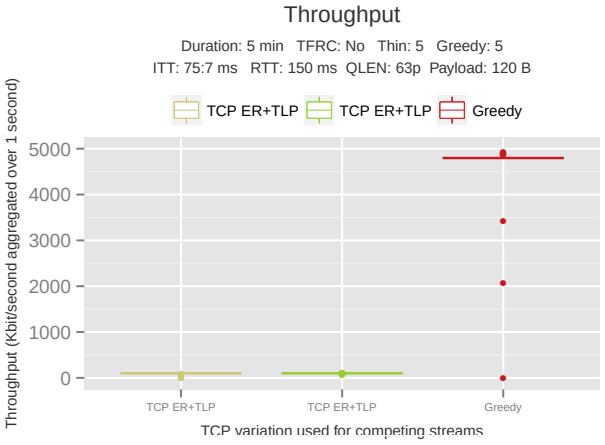
(c)

Figure A.2.15



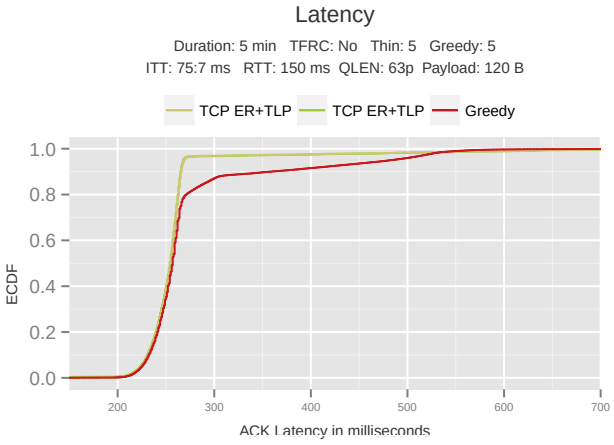
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 75	120/120/120	8/1/8/8/8/10
TCP ER+TLP	rdbsender	5	74 / 74 / 74	120/120/120	8/1/8/8/8/10
Greedy	zsender	5	12 / 12 / 13	1446/1447/1448	21/4/17/20/24/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.3 K	116	118	0	2.4 M	2.4 M	13.9 K	13.9 K	0.57 / 0.57	581
20.0 K	128	131	0	2.4 M	2.4 M	15.7 K	15.7 K	0.64 / 0.64	596
121 K	1029	1031	0	174 M	173 M	1.5 M	1.5 M	0.85 / 0.85	14527

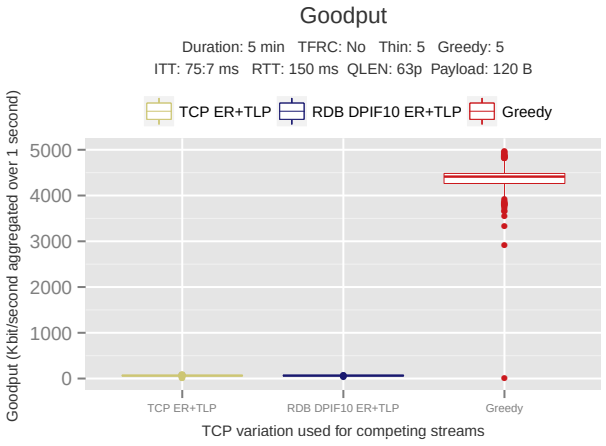
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
259	254	258	260	263	266	267	269	271	310	464	611	714	826	1152
259	254	257	260	263	265	267	268	270	330	476	617	713	748	790
277	257	260	264	271	359	436	483	500	515	527	551	725	862	1152

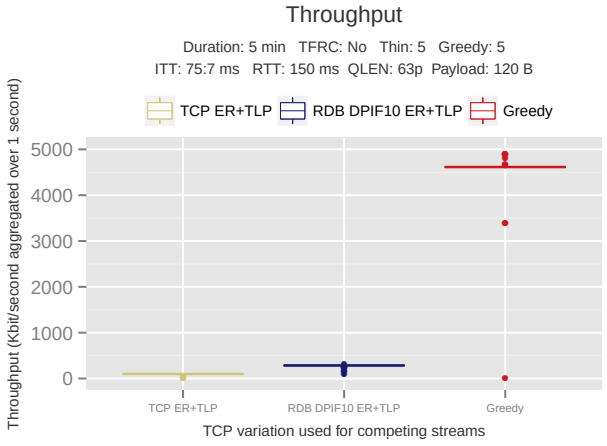
(c)

Figure A.2.16



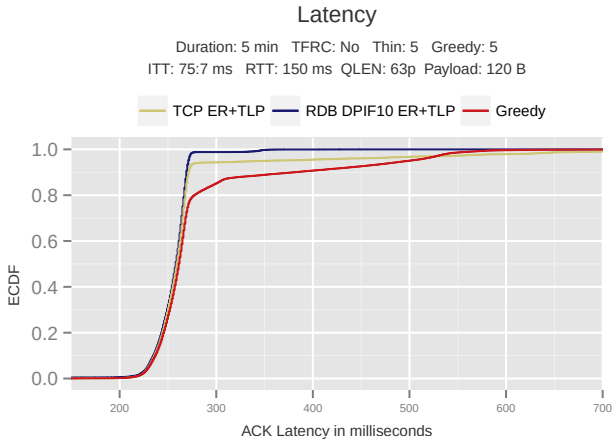
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 74	120/120/121	7/1/8/8/8/10
RDB DPIF10 ER+TLP	rdbsender	5	75 / 75 / 75	472/473/475	7/1/7/8/8/10
Greedy	zsender	5	12 / 13 / 13	1445/1447/1447	20/9/16/19/23/53

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.3 K	220	222	0	2.5 M	2.4 M	26.4 K	26.4 K	1.08 / 1.08	1041
19.8 K	232	3	19.8 K	9.4 M	2.4 M	7.0 M	360	1.19 / 1.17	
116 K	1069	1071	0	168 M	167 M	1.5 M	1.5 M	0.92 / 0.92	14850

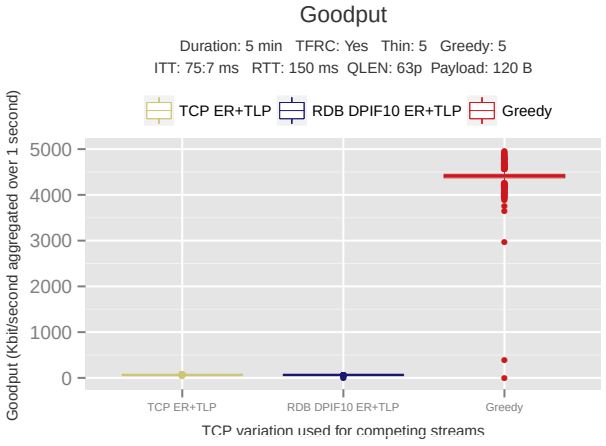
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
272	259	263	265	268	271	274	352	437	524	614	695	857	1152	1225
256	258	262	264	267	269	270	271	272	273	274	334	360	1000	1180
284	261	265	268	277	381	459	499	516	527	538	563	695	879	1152

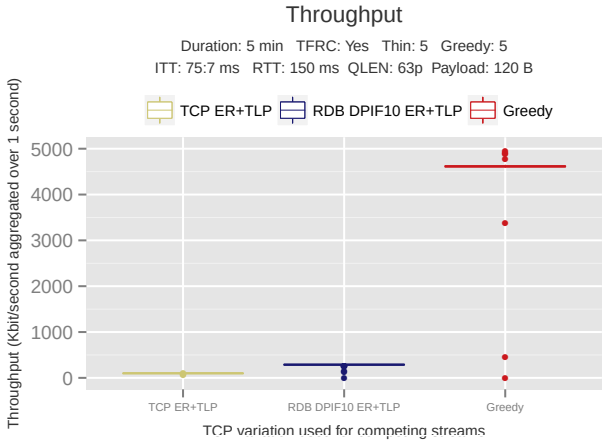
(c)

Figure A.2.17



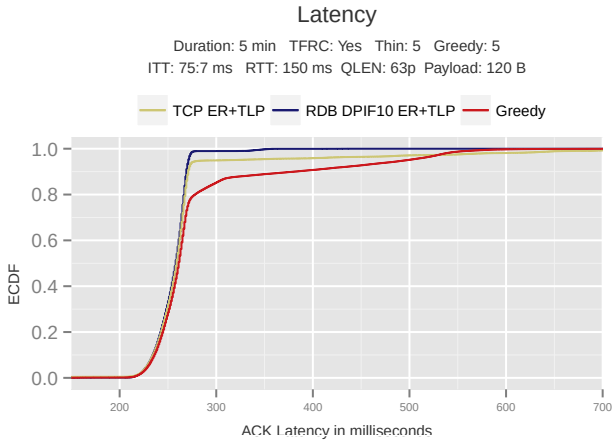
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 74	120/120/121	7/1/8/8/8/10
RDB DPIF10 ER+TLP	rdbsender	5	75 / 75 / 75	474/475/475	14/6/9/11/14/96
Greedy	zsender	5	12 / 13 / 13	1447/1447/1448	21/7/17/20/24/50

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.2 K	199	200	0	2.4 M	2.4 M	24.4 K	24.4 K	0.99 / 0.98	936
19.8 K	200	2	19.8 K	9.4 M	2.4 M	7.0 M	0	1.02 / 1.01	
116 K	1075	1077	0	168 M	166 M	1.6 M	1.6 M	0.93 / 0.93	14871

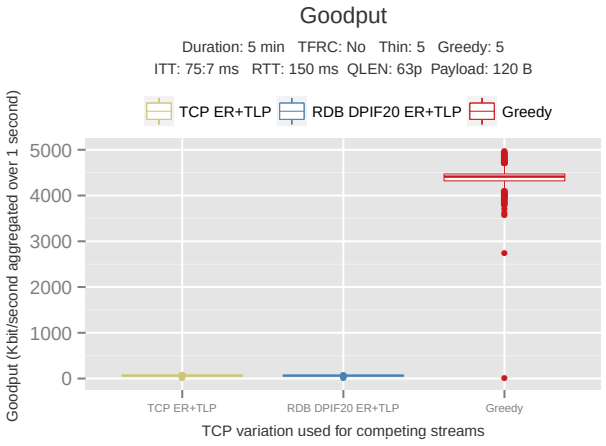
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
269	259	262	265	268	271	273	311	411	493	579	666	756	994	1070
256	258	261	264	267	269	270	271	272	273	274	323	357	450	1151
283	261	265	268	278	381	455	497	514	527	537	559	663	821	1152

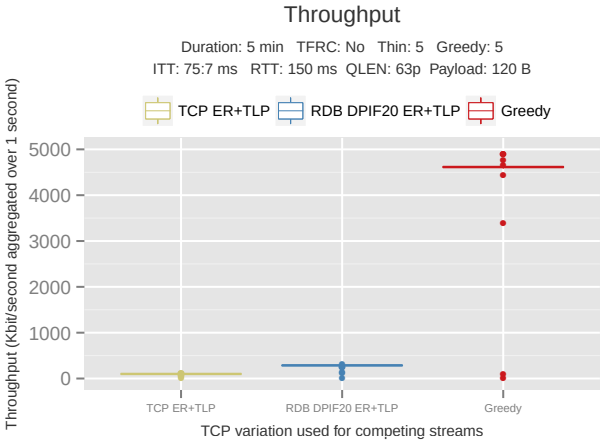
(c)

Figure A.2.18



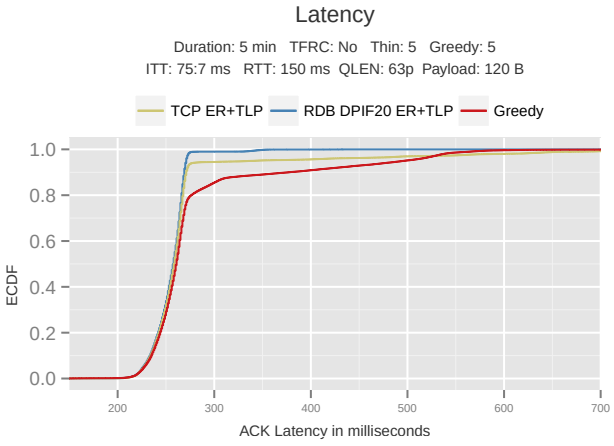
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 74	120/120/121	7/1/8/8/8/10
RDB DPIF20 ER+TLP	rdbsender	5	75 / 75 / 75	473/473/474	7/2/8/8/8/10
Greedy	zsender	5	12 / 13 / 14	1446/1447/1447	21/7/16/20/24/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.3 K	209	211	0	2.4 M	2.4 M	25.1 K	25.1 K	1.03 / 1.03	1021
19.8 K	194	2	19.8 K	9.4 M	2.4 M	7.0 M	0	0.99 / 0.98	
116 K	1070	1072	0	168 M	166 M	1.5 M	1.5 M	0.92 / 0.92	14578

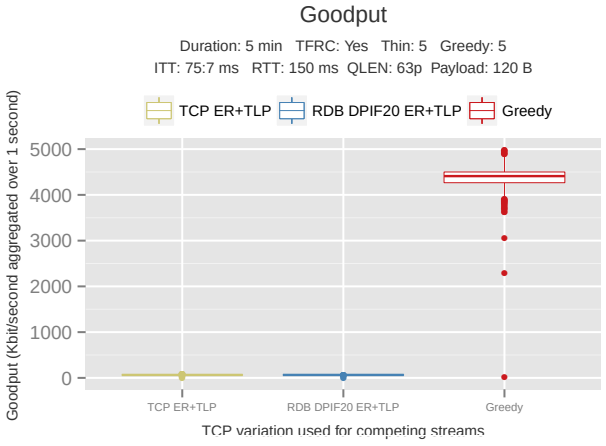
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
270	259	262	265	268	271	273	341	422	505	592	680	750	1006	1151
255	258	261	264	266	269	270	271	271	272	273	299	357	482	1150
283	261	264	268	276	377	452	496	515	527	537	565	720	812	1151

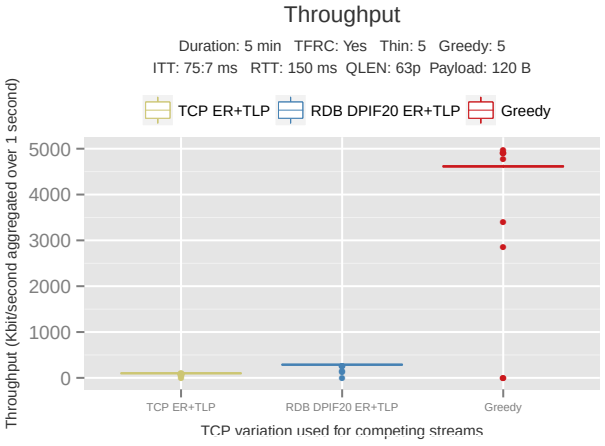
(c)

Figure A.2.19



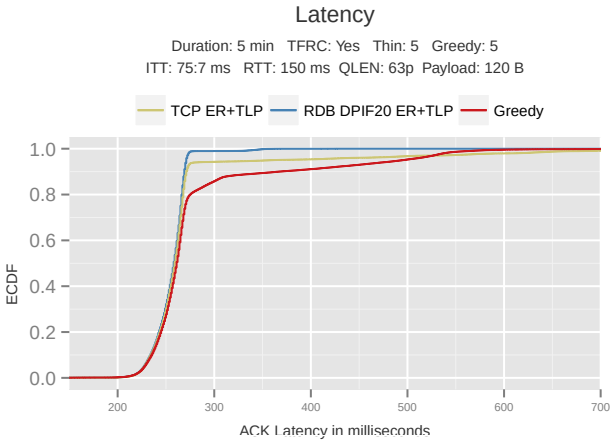
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 74	120/120/121	7/1/8/8/8/10
RDB DPIF20 ER+TLP	rdbsender	5	75 / 75 / 75	474/475/476	15/5/9/11/16/85
Greedy	zsender	5	12 / 13 / 14	1446/1447/1447	20/7/16/20/24/47

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.3 K	220	222	0	2.4 M	2.4 M	26.4 K	26.4 K	1.08 / 1.08	1072
19.9 K	198	2	19.9 K	9.4 M	2.4 M	7.1 M	0	1.01 / 1.00	
116 K	1034	1036	0	168 M	166 M	1.5 M	1.5 M	0.89 / 0.89	14091

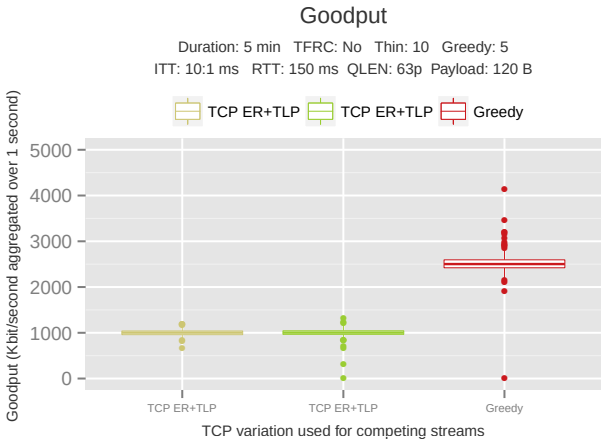
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
271	259	263	265	268	271	274	359	443	527	608	685	735	1044	1151
256	258	262	264	267	269	270	271	272	272	273	317	357	454	1152
282	261	265	268	275	367	450	495	513	527	538	564	733	860	1151

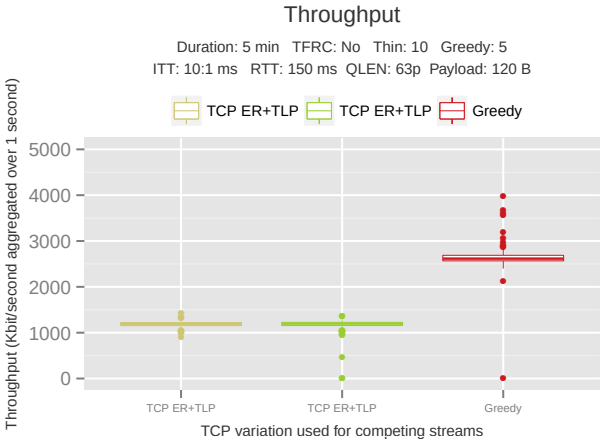
(c)

Figure A.2.20



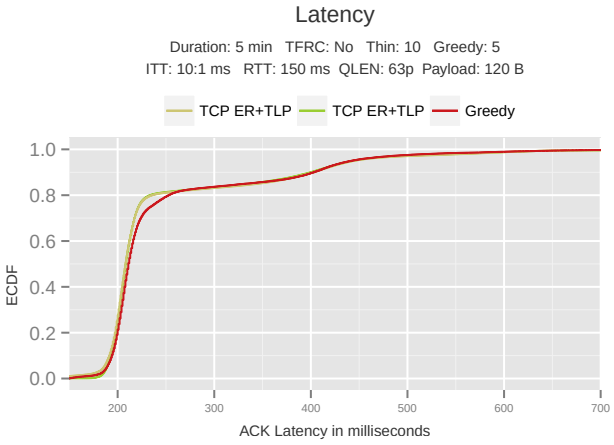
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	26 / 28 / 30	336/361/392	9/1/6/9/11/26
TCP ER+TLP	rdbsender	10	25 / 27 / 31	319/354/400	9/1/6/9/12/26
Greedy	zsender	5	21 / 22 / 25	1446/1447/1447	10/1/8/9/11/36

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
107 K	2923	2923	0	38.7 M	37.7 M	961 K	961 K	2.49 / 2.73	19280
108 K	2861	2868	0	38.1 M	37.1 M	975 K	975 K	2.56 / 2.64	19340
67.5 K	1853	1855	0	97.7 M	95.1 M	2.7 M	2.7 M	2.74 / 2.74	11064

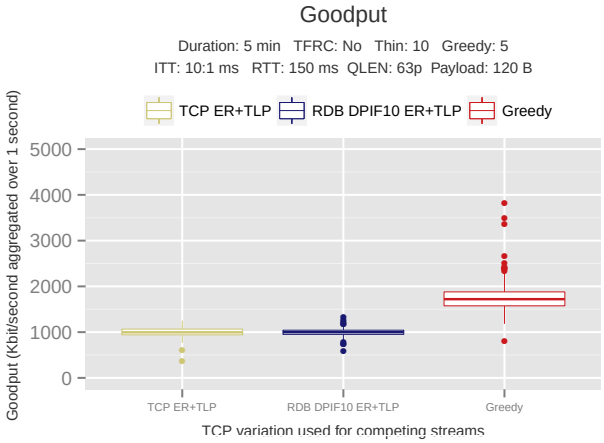
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	208	212	217	236	403	425	445	461	496	560	613	1037	1692	3875
246	209	212	217	234	400	424	444	459	487	551	612	1026	1457	2490
248	211	216	224	253	403	423	442	457	480	523	603	848	1339	1724

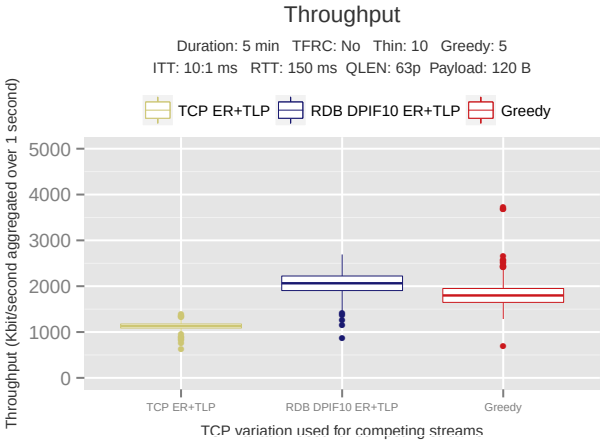
(c)

Figure A.2.21



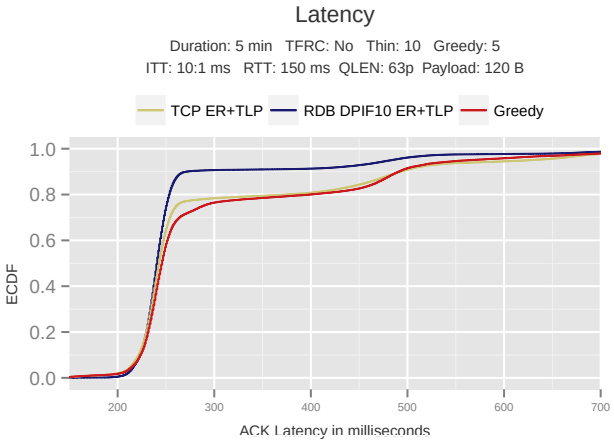
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	36 / 40 / 46	470/523/604	7/1/5/7/9/23
RDB DPIF10 ER+TLP	rdbsender	10	37 / 41 / 46	1030/1050/1076	7/1/5/7/10/25
Greedy	zsender	5	29 / 31 / 33	1446/1446/1447	9/1/6/8/10/45

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
75.0 K	3068	3068	0	39.0 M	37.7 M	1.3 M	1.3 M	3.39 / 4.09	14546
71.5 K	3010	1355	43.0 K	75.0 M	37.3 M	37.7 M	1.2 M	4.22 / 4.21	5786
47.7 K	1900	1900	0	69.0 M	66.3 M	2.7 M	2.7 M	3.98 / 3.98	9540

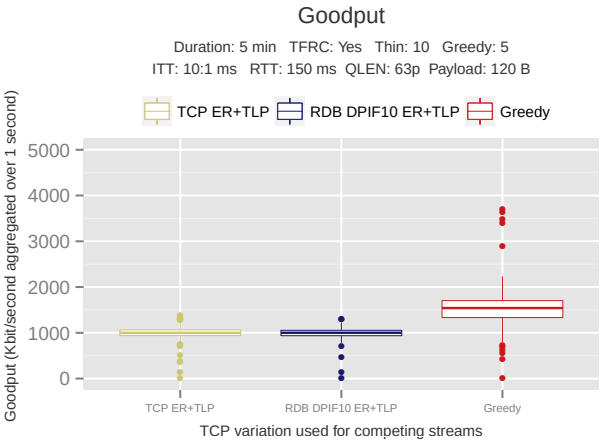
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
302	243	248	254	375	493	528	627	659	680	711	889	1440	2350	3577
269	241	244	248	253	271	453	484	498	521	659	728	1396	2129	2576
312	246	252	264	399	491	517	565	607	656	703	748	3553	11055	12510

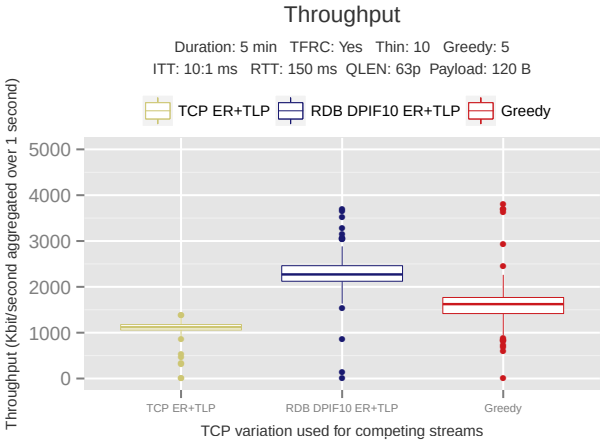
(c)

Figure A.2.22



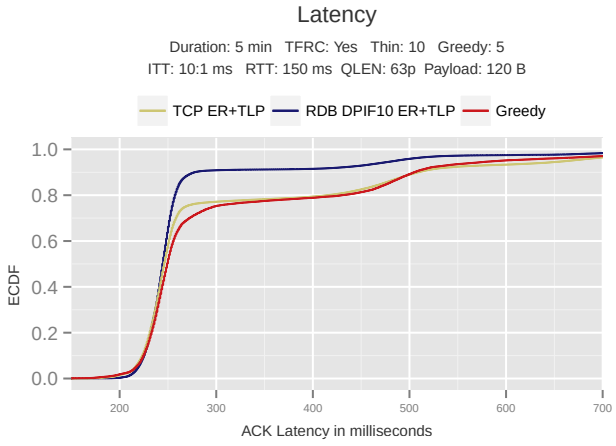
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	39 / 44 / 46	509/571/603	7/1/5/7/9/23
RDB DPIF10 ER+TLP	rdbsender	10	34 / 37 / 40	1045/1060/1074	8/1/6/8/10/101
Greedy	zsender	5	34 / 35 / 37	1445/1446/1446	8/1/6/7/9/32

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
68.8 K	3241	3248	0	39.2 M	37.5 M	1.6 M	1.6 M	4.19 / 4.71	13495
79.4 K	3644	1370	53.5 K	84.2 M	37.1 M	47.1 M	1.2 M	4.57 / 4.59	6119
42.3 K	1943	1945	0	61.1 M	58.3 M	2.8 M	2.8 M	4.59 / 4.60	8562

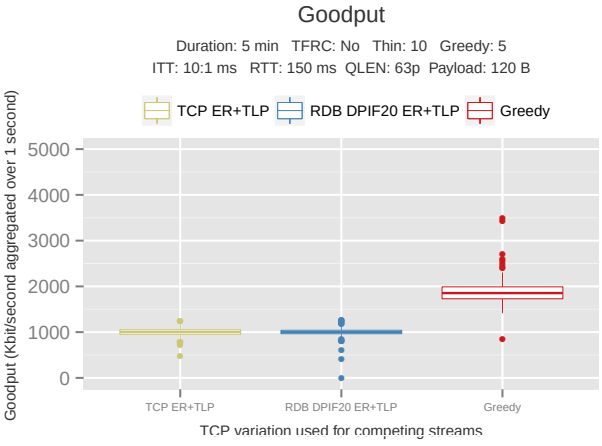
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
320	246	251	258	416	509	577	665	688	713	764	1009	4001	7287	10630
274	244	248	252	257	278	452	486	503	531	676	783	1849	2641	3041
316	249	256	272	430	505	538	593	644	697	732	899	1709	6171	6431

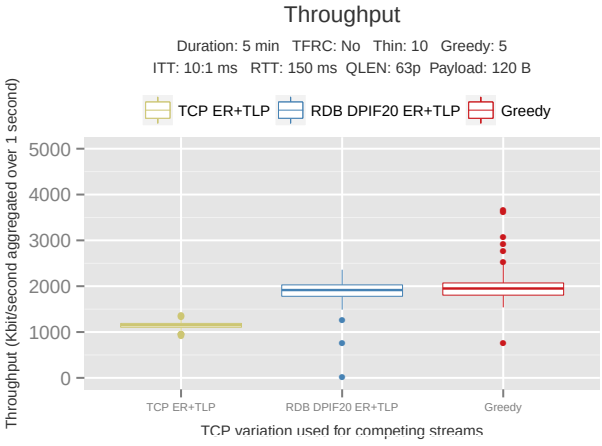
(c)

Figure A.2.23



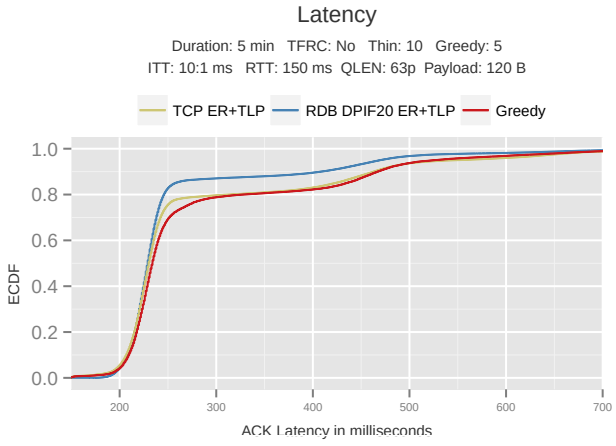
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	34 / 35 / 36	438/454/469	8/1/5/7/10/27
RDB DPIF20 ER+TLP	rdbsender	10	33 / 35 / 38	785/815/865	8/1/6/8/10/22
Greedy	zsender	5	28 / 29 / 32	1446/1447/1447	9/1/6/8/10/39

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
85.6 K	3145	3145	0	38.8 M	37.7 M	1.1 M	1.1 M	2.88 / 3.67	16331
83.5 K	3093	1781	35.4 K	68.0 M	37.1 M	30.8 M	999 K	3.71 / 3.70	10665
51.0 K	2043	2043	0	73.8 M	70.8 M	3.0 M	3.0 M	4.00 / 4.00	9524

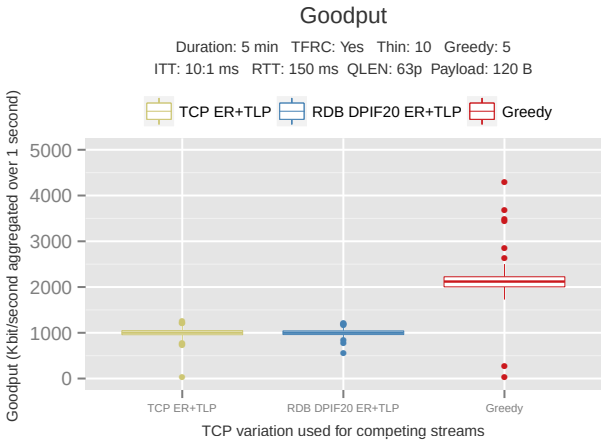
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
281	230	235	242	314	463	490	548	601	636	664	720	1195	1675	1910
260	229	233	238	246	408	447	470	484	507	579	669	1147	1900	2942
285	234	240	251	328	467	491	526	559	605	654	706	1159	1841	2367

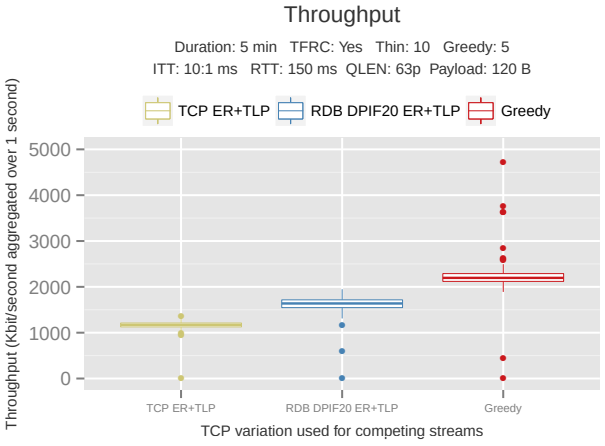
(c)

Figure A.2.24



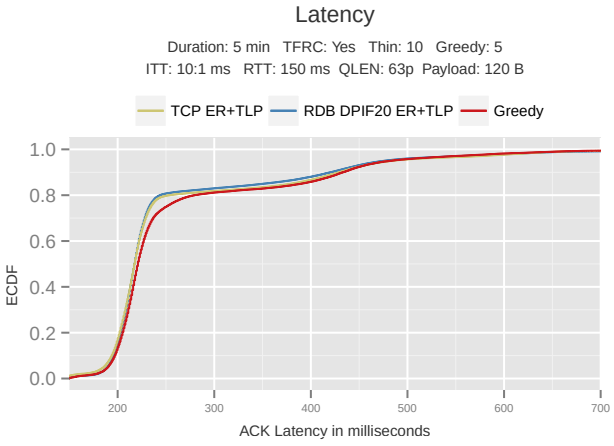
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	28 / 30 / 33	367/396/435	9/1/6/8/11/25
RDB DPIF20 ER+TLP	rdbsender	10	28 / 31 / 34	538/589/637	10/1/6/9/13/25
Greedy	zsender	5	25 / 26 / 28	1446/1447/1447	9/1/7/8/10/26

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
98.2 K	3135	3135	0	38.8 M	37.7 M	1.1 M	1.1 M	2.72 / 3.19	17763
96.0 K	3318	2541	22.9 K	56.4 M	37.2 M	19.2 M	1.0 M	3.39 / 3.46	16388
57.5 K	2071	2073	0	83.2 M	80.2 M	3.0 M	3.0 M	3.60 / 3.60	10031

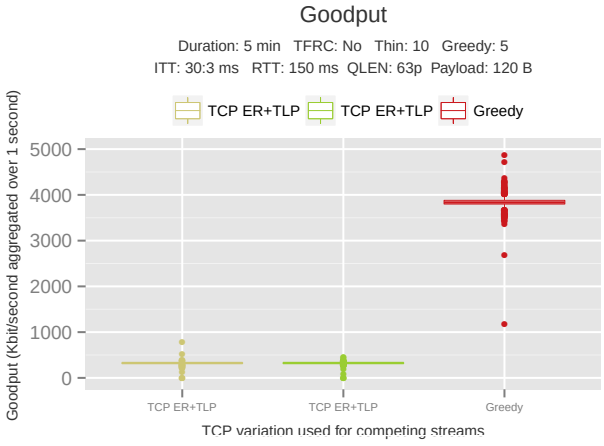
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
259	217	222	229	254	429	456	484	514	571	612	656	1091	1513	1734
257	217	222	228	244	420	449	476	500	550	606	673	1124	1601	2348
264	221	227	237	280	433	456	482	507	546	591	653	887	1289	1499

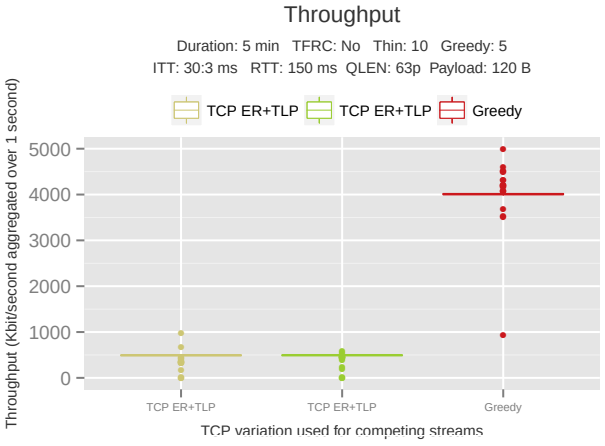
(c)

Figure A.2.25



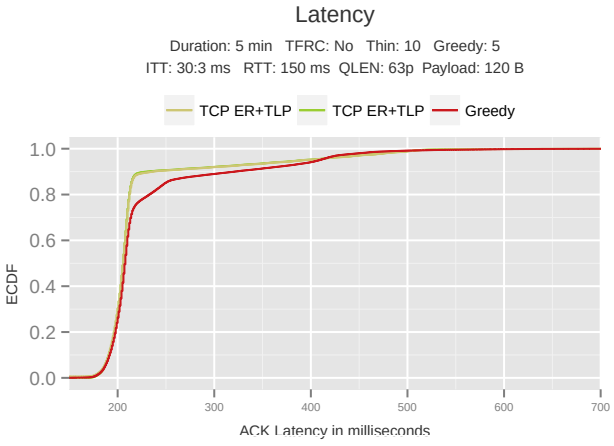
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	31 / 31 / 32	127/128/131	10/1/8/11/12/13
TCP ER+TLP	rdbsender	10	31 / 31 / 32	126/127/131	10/1/8/11/12/14
Greedy	zsender	5	14 / 15 / 15	1447/1447/1447	14/5/11/14/16/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
95.8 K	1232	1234	0	12.3 M	12.2 M	149 K	149 K	1.21 / 1.29	8927
94.9 K	1197	1204	0	12.1 M	11.9 M	148 K	148 K	1.22 / 1.26	8705
102 K	1369	1369	0	147 M	145 M	2.0 M	2.0 M	1.35 / 1.35	13025

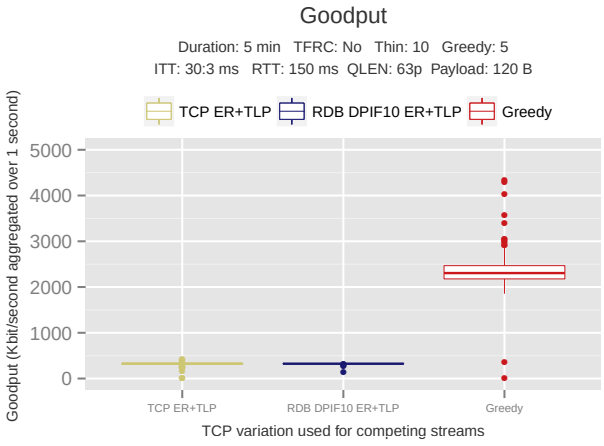
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
223	206	208	210	212	237	334	393	420	449	476	504	681	3272	7739
222	206	208	210	212	230	330	391	419	447	476	503	621	1235	7271
231	208	210	214	233	321	383	409	417	426	450	488	643	892	999

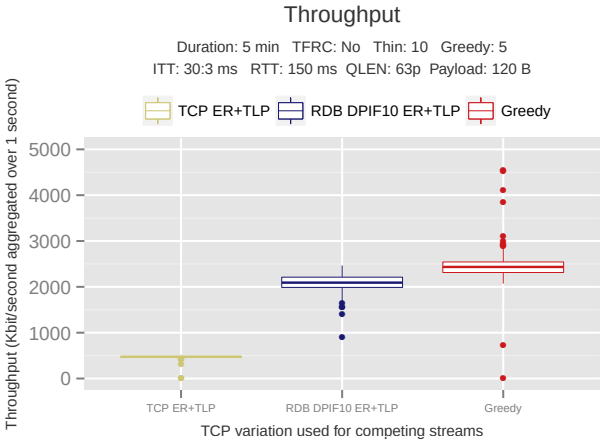
(c)

Figure A.2.26



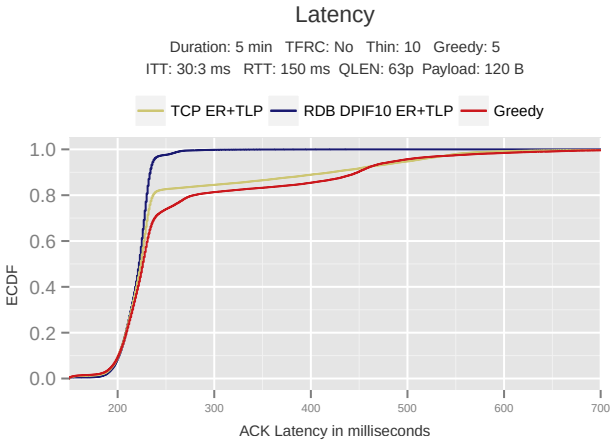
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	34 / 34 / 35	139/143/148	8/1/6/8/11/13
RDB DPIF10 ER+TLP	rdbsender	10	36 / 37 / 39	927/929/931	8/1/6/8/10/14
Greedy	zsender	5	22 / 24 / 25	1446/1447/1448	10/2/8/9/11/31

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
87.2 K	2165	2172	0	12.5 M	12.2 M	280 K	280 K	2.24 / 2.48	13413
80.3 K	2075	2	80.3 K	74.6 M	12.0 M	62.6 M	1.1 K	2.68 / 2.58	2
63.2 K	1994	1996	0	91.4 M	88.5 M	2.9 M	2.9 M	3.15 / 3.16	10922

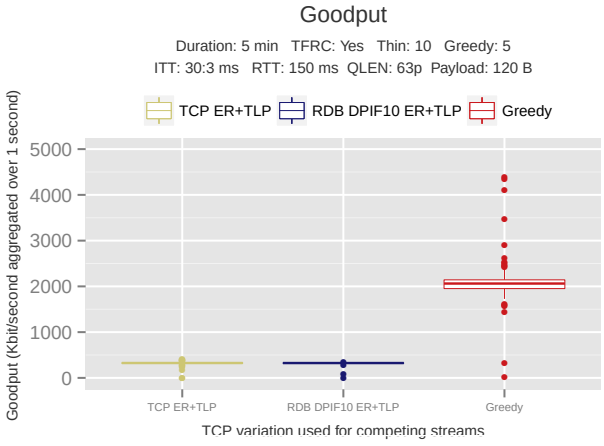
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
257	224	227	231	237	421	470	504	519	535	551	587	912	1210	1291
221	222	225	228	230	234	235	237	239	242	255	263	333	436	1157
267	226	231	238	279	448	466	490	506	531	573	641	913	1193	1202

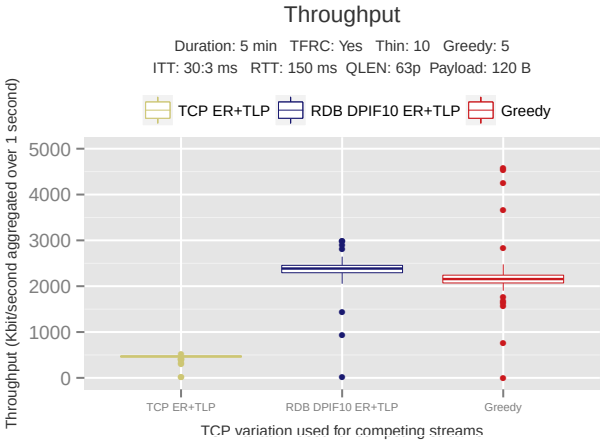
(c)

Figure A.2.27



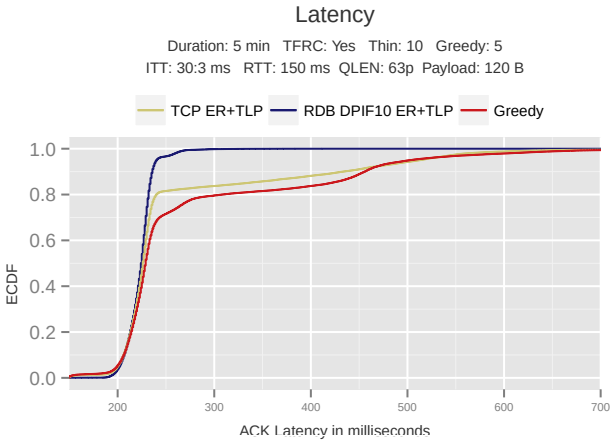
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	35 / 36 / 39	146/151/162	8/1/6/8/11/13
RDB DPIF10 ER+TLP	rdbsender	10	32 / 33 / 34	946/948/951	9/2/7/8/11/59
Greedy	zsender	5	24 / 27 / 28	1447/1447/1447	9/3/7/8/10/34

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
82.8 K	2234	2241	0	12.5 M	12.2 M	317 K	317 K	2.53 / 2.70	13302
89.5 K	2374	7	89.5 K	84.9 M	12.0 M	72.9 M	0	2.74 / 2.65	
56.2 K	1991	1993	0	81.4 M	78.5 M	2.9 M	2.9 M	3.54 / 3.54	10483

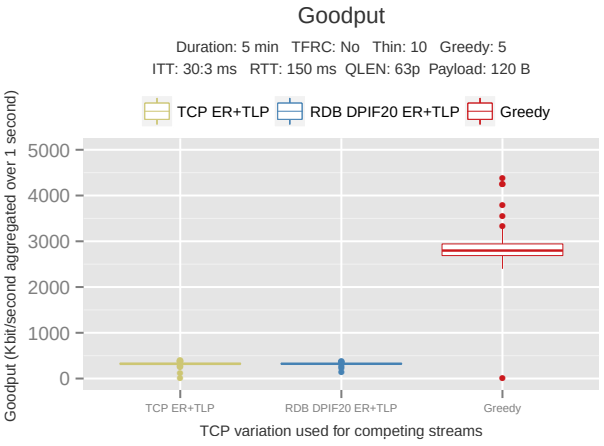
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
263	226	229	233	240	433	480	513	527	543	562	616	1084	2628	2772
224	224	227	229	232	236	238	240	243	254	260	266	323	405	1152
276	229	233	243	308	456	475	503	524	555	605	661	1035	1750	1866

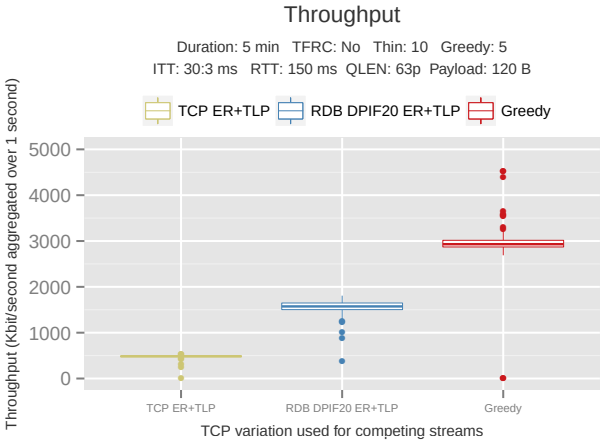
(c)

Figure A.2.28



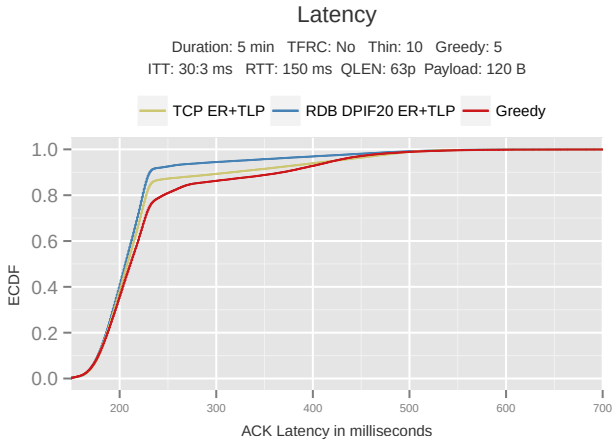
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	31 / 32 / 33	130/133/135	9/1/7/9/11/13
RDB DPIF20 ER+TLP	rdbsender	10	31 / 32 / 33	561/581/597	9/1/7/9/11/13
Greedy	zsender	5	19 / 20 / 21	1447/1447/1448	10/4/8/10/11/39

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
93.2 K	1770	1777	0	12.4 M	12.2 M	223 K	223 K	1.80 / 1.90	12119
92.0 K	1773	823	59.5 K	53.5 M	12.0 M	41.4 M	99.1 K	1.89 / 1.93	6534
75.4 K	1791	1793	0	109 M	107 M	2.6 M	2.6 M	2.37 / 2.37	11566

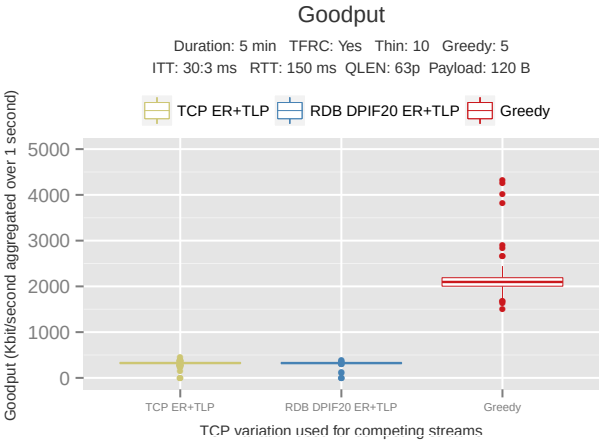
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
228	208	215	221	227	316	381	422	442	461	480	504	624	1028	1152
216	206	212	218	224	231	256	321	362	403	446	491	588	953	1095
235	211	219	226	245	368	402	422	434	448	469	505	643	999	1152

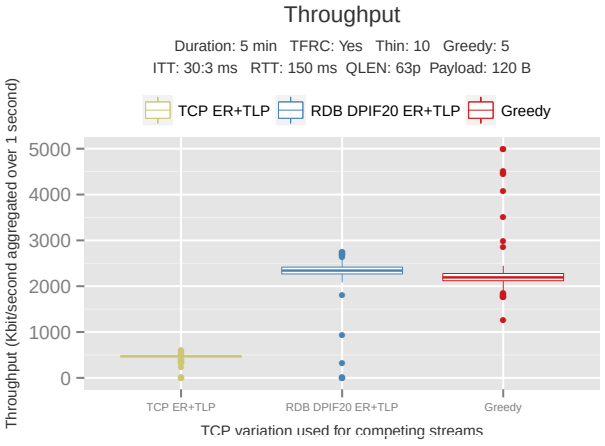
(c)

Figure A.2.29



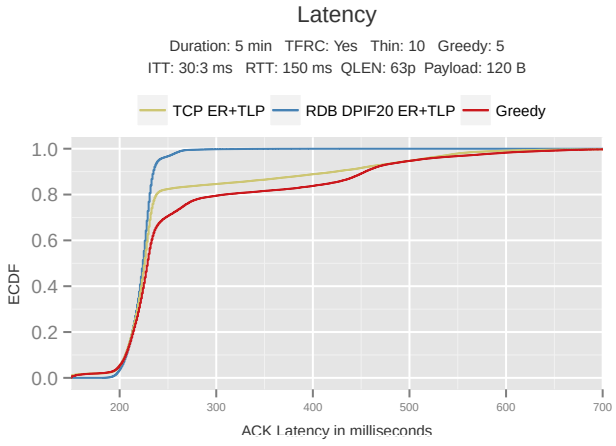
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	35 / 36 / 37	143/148/153	8/1/6/8/11/13
RDB DPIF20 ER+TLP	rdbsender	10	33 / 34 / 35	943/945/948	9/1/6/8/11/135
Greedy	zsender	5	24 / 26 / 27	1446/1447/1447	9/2/7/8/10/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
83.8 K	2073	2080	0	12.4 M	12.2 M	282 K	282 K	2.27 / 2.47	12778
87.5 K	2322	14	87.3 K	82.7 M	11.9 M	70.8 M	120	2.70 / 2.65	8
57.8 K	1981	1981	0	83.6 M	80.8 M	2.9 M	2.9 M	3.43 / 3.43	10732

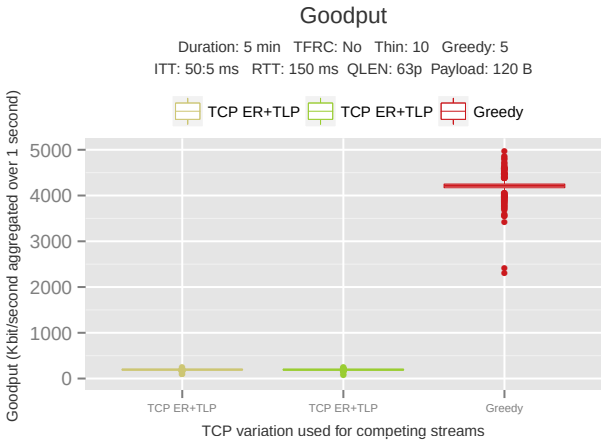
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
259	226	229	232	238	423	472	505	520	536	552	589	820	1813	7449
224	224	227	229	231	235	237	241	245	252	257	263	338	999	4005
275	229	233	248	309	454	475	506	526	558	591	636	768	999	1152

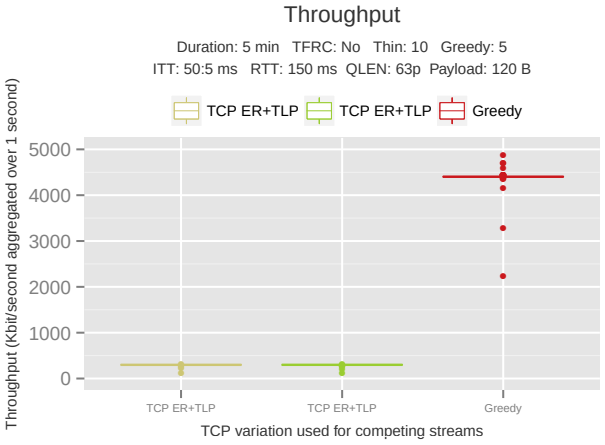
(c)

Figure A.2.30



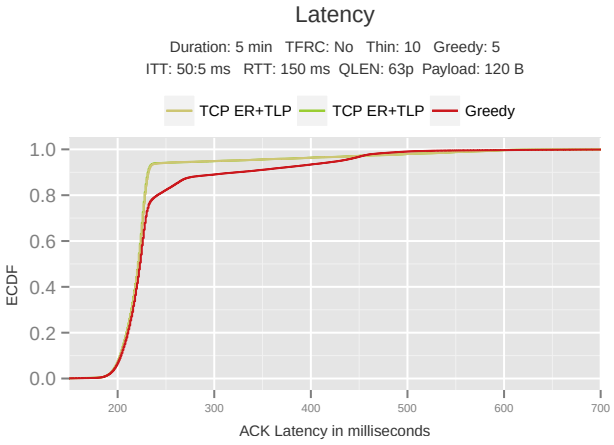
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	49 / 50 / 50	121/122/122	8/1/8/9/9/10
TCP ER+TLP	rdbsender	10	50 / 50 / 50	121/122/122	8/1/8/9/9/10
Greedy	zsender	5	13 / 13 / 14	1446/1447/1447	17/1/13/16/19/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
60.2 K	560	560	0	7.3 M	7.3 M	67.4 K	67.4 K	0.92 / 0.93	3166
59.6 K	546	546	0	7.3 M	7.2 M	66.0 K	66.0 K	0.91 / 0.92	3219
111 K	1202	1202	0	160 M	159 M	1.7 M	1.7 M	1.08 / 1.08	13671

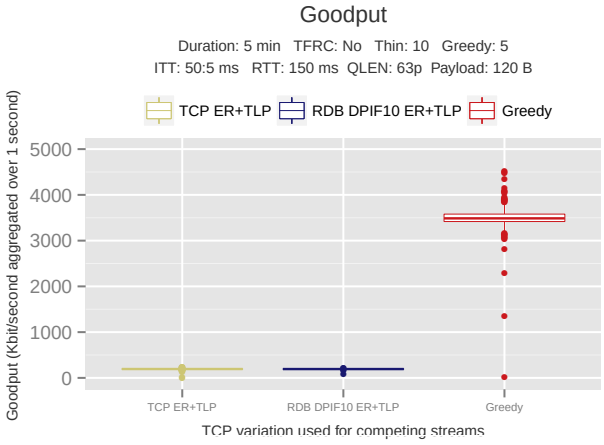
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
232	221	224	226	228	232	235	306	383	443	502	561	626	683	805
231	221	224	226	228	232	234	310	380	443	502	561	639	900	974
245	223	226	229	241	324	392	430	443	452	465	497	702	2770	2788

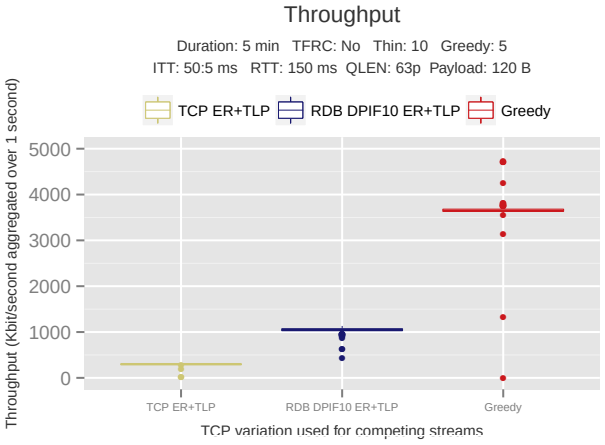
(c)

Figure A.2.31



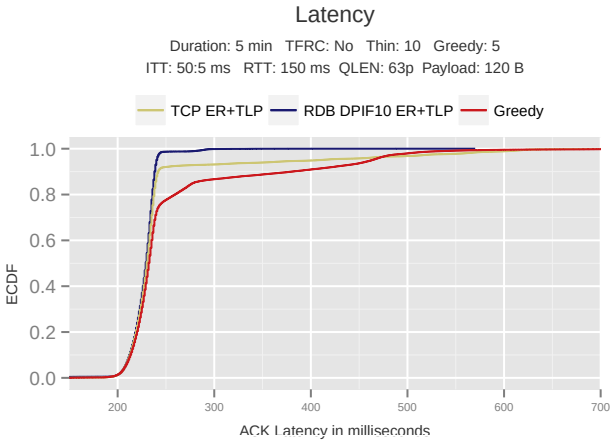
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 50 / 51	122/123/124	8/1/8/9/9/10
RDB DPIF10 ER+TLP	rdbsender	10	50 / 51 / 51	603/604/605	8/2/7/9/9/10
Greedy	zsender	5	15 / 16 / 17	1446/1447/1448	15/3/12/14/17/40

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
60.0 K	718	724	0	7.4 M	7.3 M	86.4 K	86.4 K	1.17 / 1.20	4004
58.6 K	711	1	58.6 K	35.4 M	7.2 M	28.2 M	120	1.24 / 1.21	1
92.7 K	1299	1301	0	134 M	132 M	1.9 M	1.9 M	1.40 / 1.40	13053

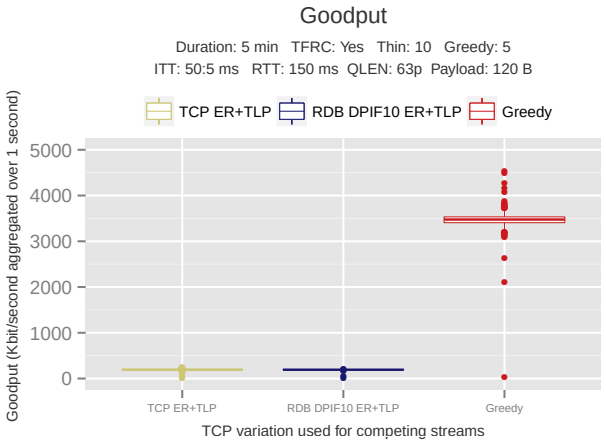
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	230	233	235	238	242	291	407	460	510	559	606	708	1150	1367
228	229	232	234	236	238	240	240	241	242	244	280	318	396	539
259	232	235	239	259	380	441	463	471	480	501	538	741	973	1153

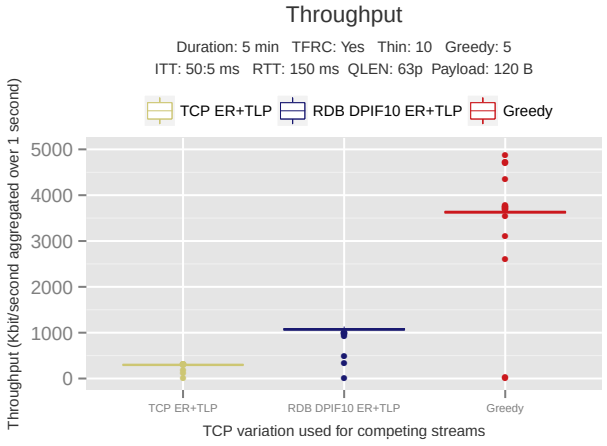
(c)

Figure A.2.32



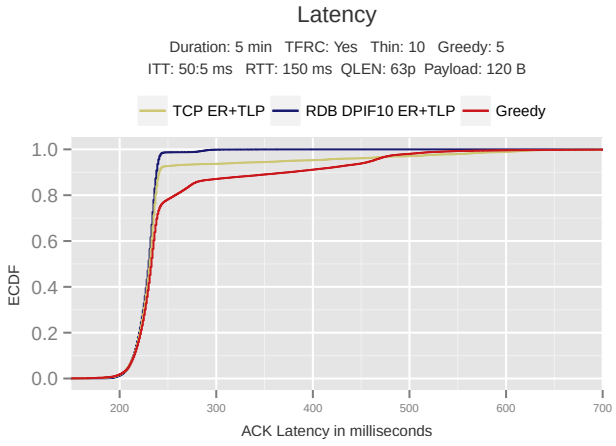
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 50 / 50	122/122/123	8/1/8/9/10/10
RDB DPIF10 ER+TLP	rdbsender	10	50 / 50 / 50	606/607/608	11/4/9/9/10/89
Greedy	zsender	5	16 / 16 / 17	1447/1447/1448	14/4/12/14/17/40

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
59.9 K	641	648	0	7.3 M	7.3 M	77.2 K	77.2 K	1.05 / 1.07	3670
59.6 K	726	7	59.6 K	36.2 M	7.2 M	29.1 M	0	1.24 / 1.22	
92.2 K	1287	1289	0	133 M	132 M	1.9 M	1.9 M	1.40 / 1.40	12291

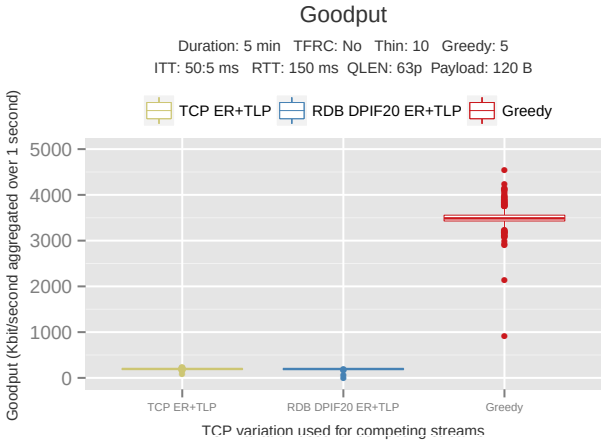
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
245	231	233	235	237	241	258	376	440	497	553	604	730	1151	1152
229	230	232	234	236	238	239	240	241	241	243	280	308	1152	1153
259	233	235	239	258	375	435	462	469	477	498	533	735	966	1151

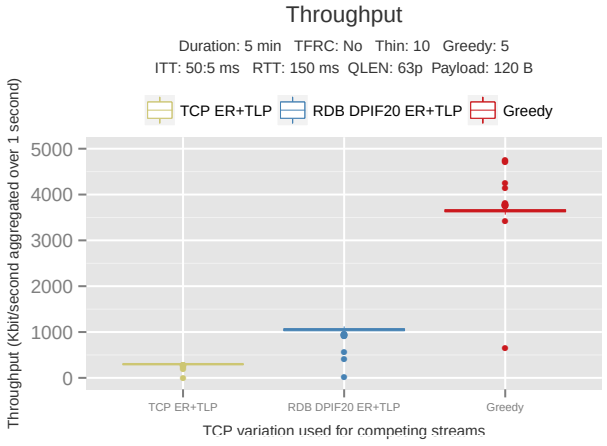
(c)

Figure A.2.33



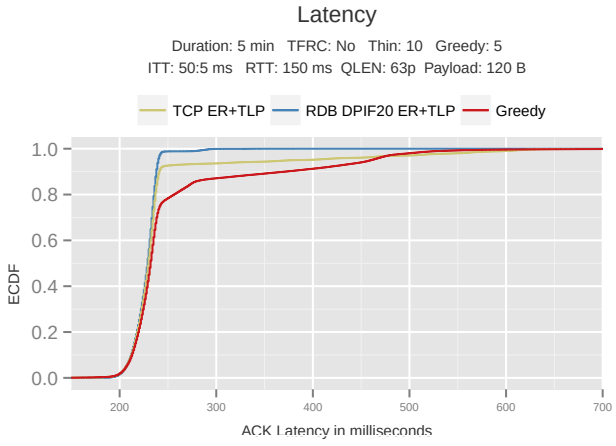
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 50 / 50	122/122/124	8/1/8/9/9/10
RDB DPIF20 ER+TLP	rdbsender	10	50 / 51 / 51	602/603/605	8/2/7/9/9/10
Greedy	zsender	5	15 / 16 / 17	1446/1447/1447	14/3/12/14/17/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
60.1 K	660	662	0	7.4 M	7.3 M	79.6 K	79.6 K	1.08 / 1.10	3779
58.6 K	656	7	58.6 K	35.4 M	7.2 M	28.2 M	0	1.15 / 1.12	
92.5 K	1261	1261	0	134 M	132 M	1.8 M	1.8 M	1.36 / 1.36	12388

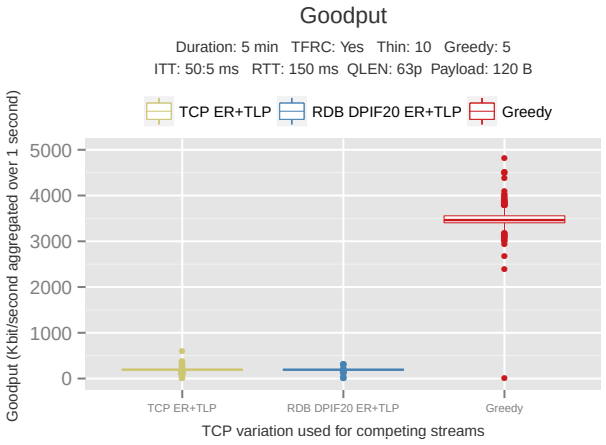
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
244	230	232	235	237	241	261	380	441	497	549	600	667	1040	1327
228	229	231	234	236	238	239	240	241	241	243	278	299	1242	1244
257	232	235	239	256	371	433	460	469	478	500	531	715	879	999

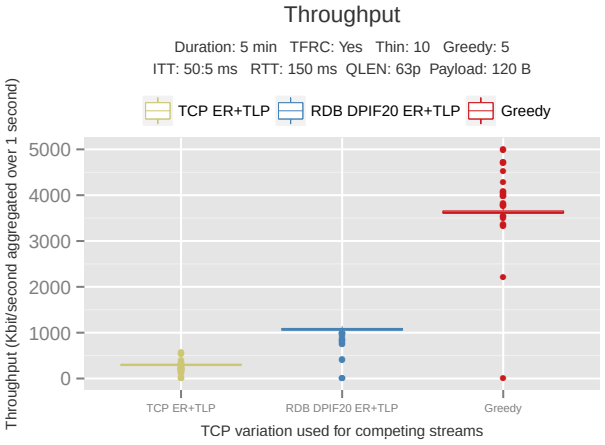
(c)

Figure A.2.34



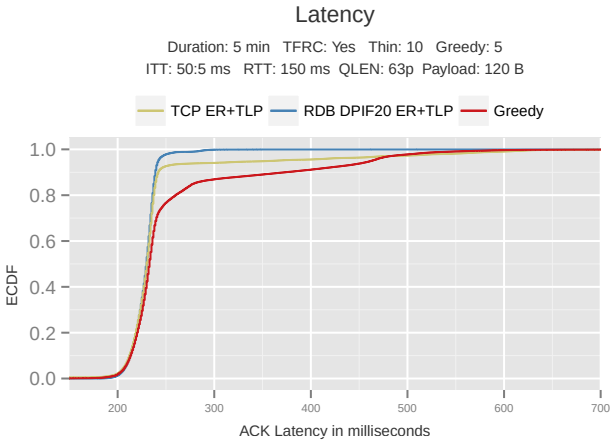
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 52 / 55	122/126/132	8/1/8/9/10/10
RDB DPIF20 ER+TLP	rdbsender	10	50 / 51 / 53	609/609/612	11/1/9/10/10/125
Greedy	zsender	5	15 / 16 / 17	1446/1447/1448	15/4/12/14/17/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
58.3 K	587	593	0	7.3 M	7.2 M	87.0 K	87.0 K	1.19 / 1.01	3360
58.6 K	636	21	58.6 K	35.8 M	7.1 M	28.6 M	9.3 K	1.11 / 1.08	13
92.5 K	1253	1253	0	134 M	132 M	1.8 M	1.8 M	1.35 / 1.35	12503

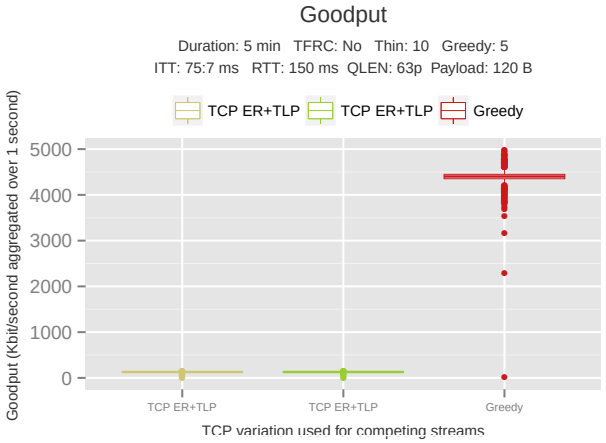
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	230	233	235	237	242	253	361	419	475	539	597	728	10514	19860
230	230	232	234	236	239	240	242	244	246	251	279	314	3259	7760
259	233	236	240	260	374	438	463	470	481	506	545	711	992	1124

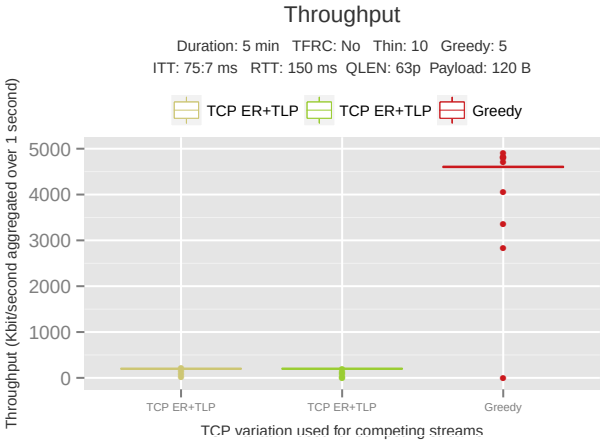
(c)

Figure A.2.35



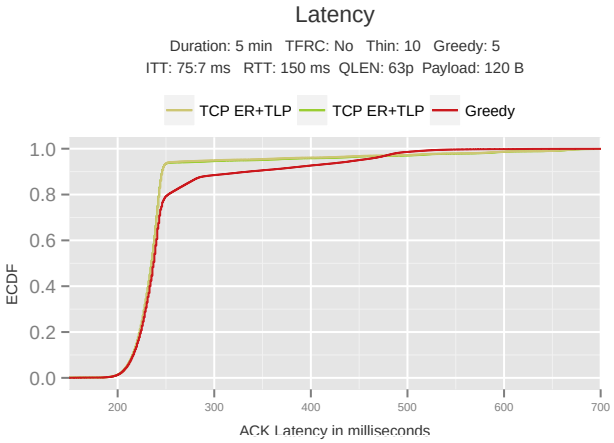
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	74 / 74 / 75	120/120/120	7/1/7/8/8/10
TCP ER+TLP	rdbsender	10	74 / 74 / 74	120/120/121	7/1/7/8/8/10
Greedy	zsender	5	11 / 13 / 14	1446/1447/1447	19/5/15/18/22/46

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.5 K	420	428	0	4.9 M	4.8 M	50.6 K	50.6 K	1.04 / 1.04	1905
40.0 K	441	453	0	4.8 M	4.8 M	53.4 K	53.4 K	1.10 / 1.10	2041
116 K	1154	1154	0	168 M	166 M	1.7 M	1.7 M	0.99 / 1.00	14086

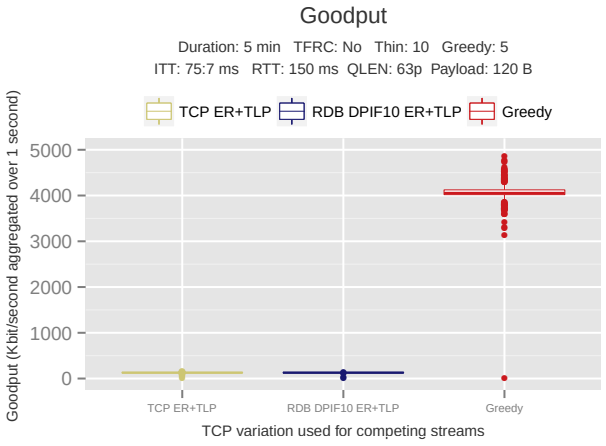
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	235	238	240	243	246	249	304	384	467	558	641	719	1151	1183
247	235	238	240	243	246	249	342	421	498	573	646	708	1152	1152
258	237	240	243	252	335	409	449	466	478	488	512	639	948	1151

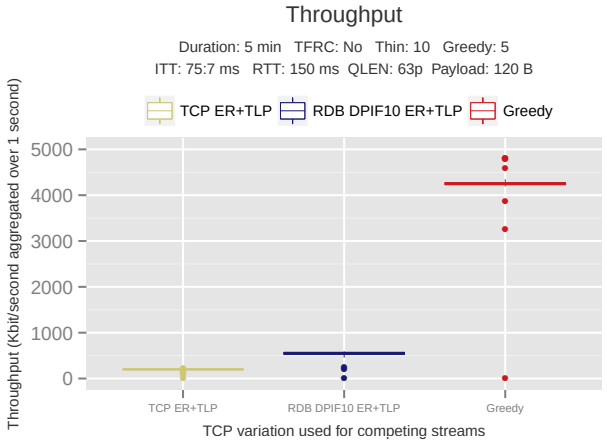
(c)

Figure A.2.36



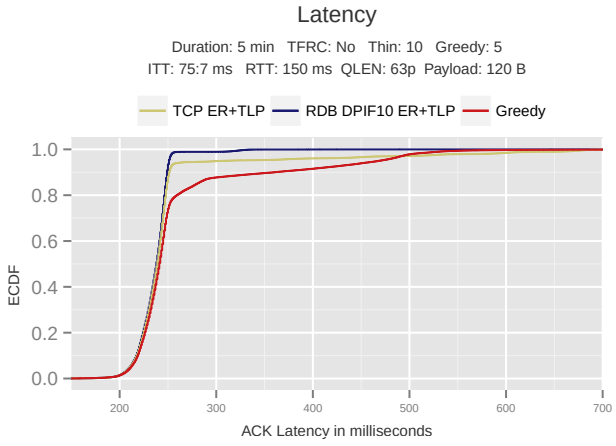
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	74 / 74 / 75	120/120/121	7/1/7/8/8/10
RDB DPIF10 ER+TLP	rdbsender	10	75 / 75 / 75	449/451/452	7/2/7/8/8/10
Greedy	zsender	5	13 / 14 / 15	1445/1447/1447	17/1/14/17/20/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.5 K	415	423	0	4.9 M	4.8 M	49.9 K	49.9 K	1.02 / 1.02	1943
39.6 K	426	7	39.6 K	17.9 M	4.8 M	13.1 M	0	1.12 / 1.07	
107 K	1189	1191	0	155 M	153 M	1.7 M	1.7 M	1.11 / 1.11	13415

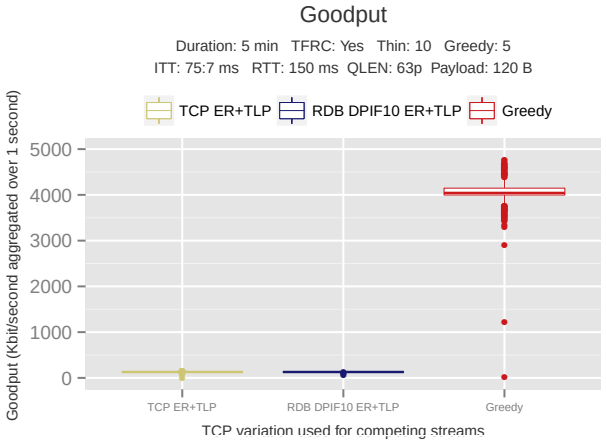
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
250	239	242	245	248	251	254	309	393	480	573	653	705	1151	1151
237	238	241	244	247	249	250	251	252	253	254	309	338	1151	1152
264	241	245	248	259	360	433	469	482	491	503	531	718	1645	1893

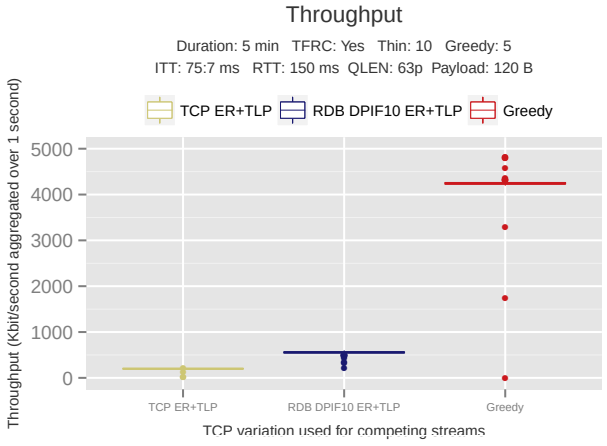
(c)

Figure A.2.37



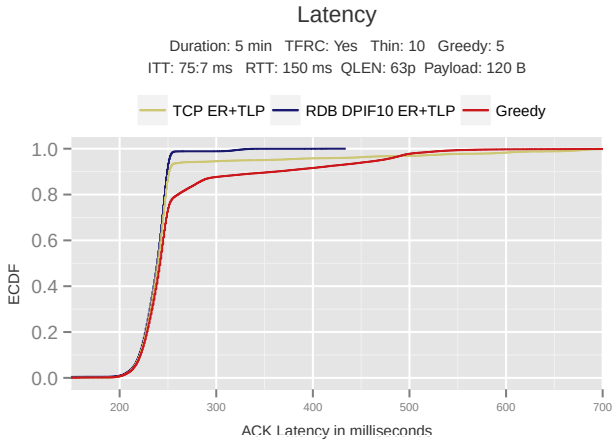
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	74 / 74 / 74	120/120/121	7/1/7/8/8/10
RDB DPIF10 ER+TLP	rdbsender	10	74 / 75 / 75	453/455/457	9/3/8/8/10/27
Greedy	zsender	5	13 / 14 / 14	1446/1447/1447	17/4/14/16/20/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.6 K	452	461	0	4.9 M	4.8 M	54.5 K	54.5 K	1.11 / 1.11	2037
39.8 K	447	4	39.8 K	18.1 M	4.8 M	13.4 M	480	1.17 / 1.12	4
107 K	1187	1189	0	155 M	153 M	1.7 M	1.7 M	1.11 / 1.11	13501

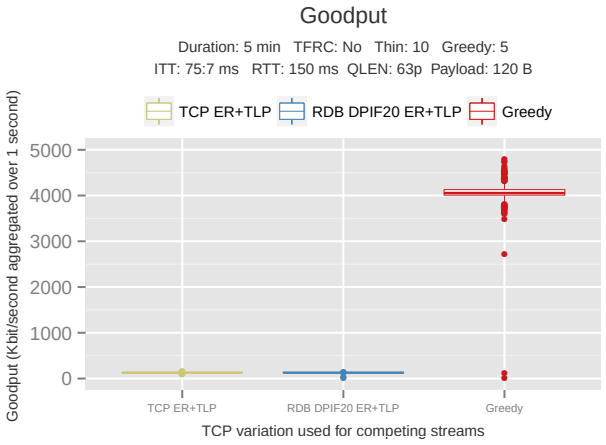
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
253	240	243	245	248	251	254	343	431	510	586	663	773	1151	1153
237	239	242	244	247	249	250	251	252	253	254	311	335	410	413
265	242	245	249	260	362	431	470	482	492	504	533	706	828	1152

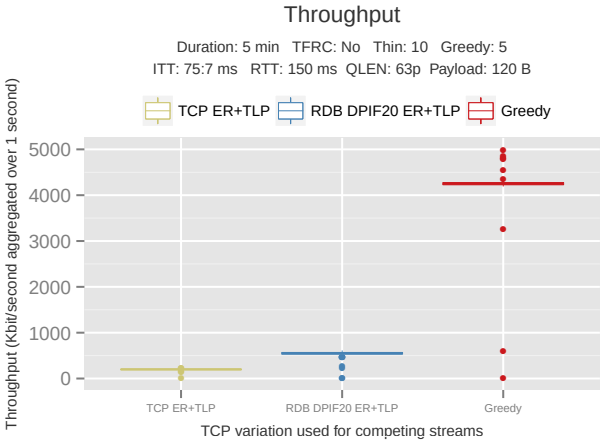
(c)

Figure A.2.38



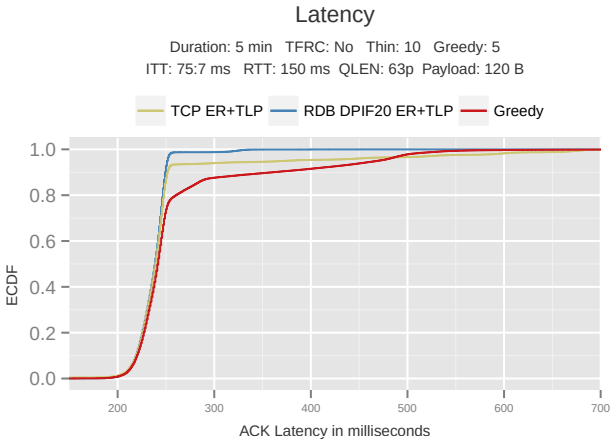
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	74 / 74 / 74	120/120/121	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	10	75 / 75 / 75	451/453/454	7/2/7/8/8/10
Greedy	zsender	5	13 / 14 / 15	1447/1447/1447	18/4/14/17/20/55

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.6 K	498	502	0	4.9 M	4.8 M	60.2 K	60.2 K	1.22 / 1.22	2268
39.6 K	478	11	39.5 K	17.9 M	4.8 M	13.2 M	240	1.26 / 1.21	2
107 K	1163	1165	0	155 M	153 M	1.7 M	1.7 M	1.09 / 1.09	13619

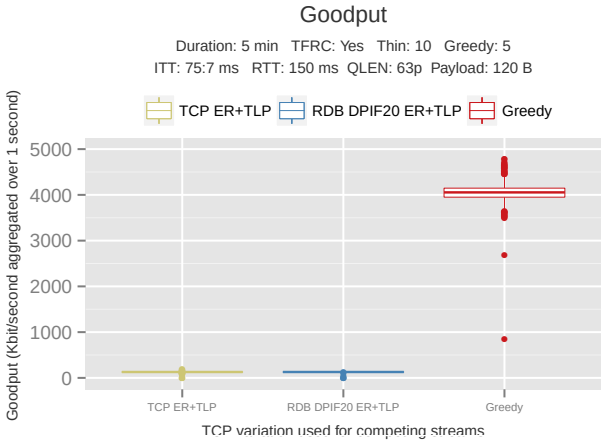
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
253	239	243	245	248	252	255	375	448	520	593	663	728	946	998
238	239	242	244	247	249	251	252	252	253	254	315	343	1151	1153
264	242	245	249	260	360	434	469	482	491	503	532	715	788	1151

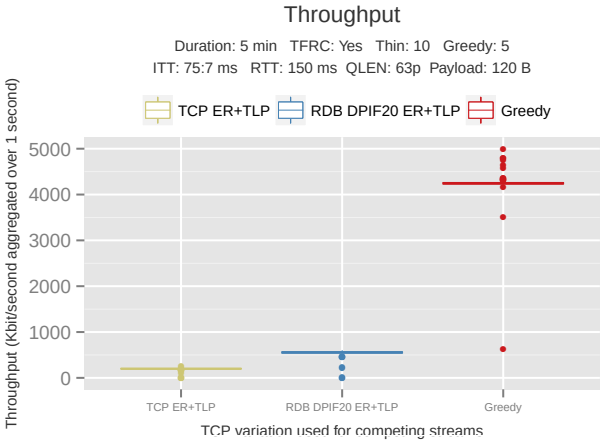
(c)

Figure A.2.39



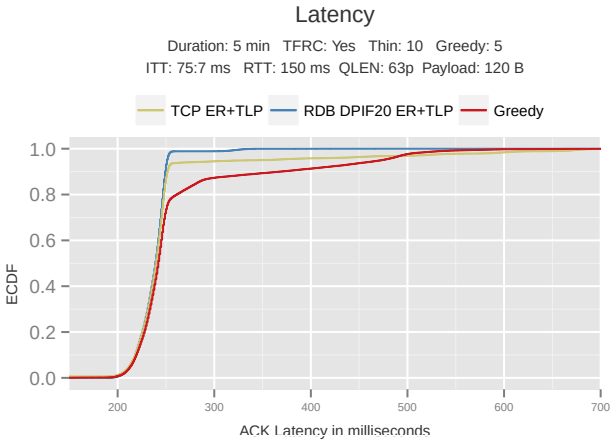
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	74 / 75 / 77	120/121/124	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	10	75 / 75 / 75	454/454/456	10/1/8/8/10/35
Greedy	zsender	5	13 / 14 / 14	1447/1447/1447	18/6/14/17/21/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.4 K	443	450	0	4.9 M	4.8 M	53.9 K	53.9 K	1.11 / 1.10	2089
39.4 K	437	14	39.4 K	17.9 M	4.7 M	13.2 M	120	1.13 / 1.11	4
107 K	1171	1171	0	155 M	153 M	1.7 M	1.7 M	1.09 / 1.09	13857

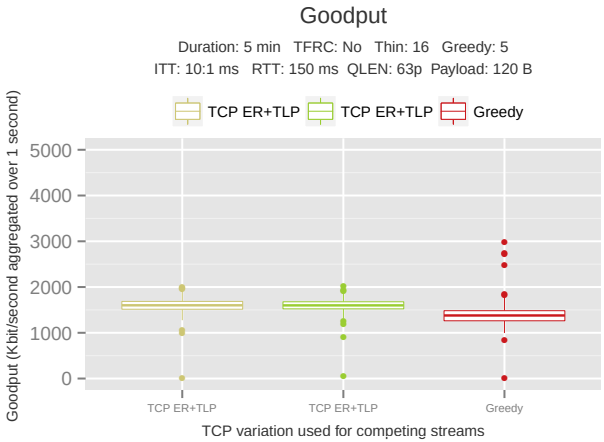
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
252	240	243	246	248	252	254	348	426	505	583	656	704	3151	7493
238	239	242	245	247	249	250	251	252	253	254	312	338	3280	3282
265	242	245	249	261	368	438	473	485	493	506	537	678	822	999

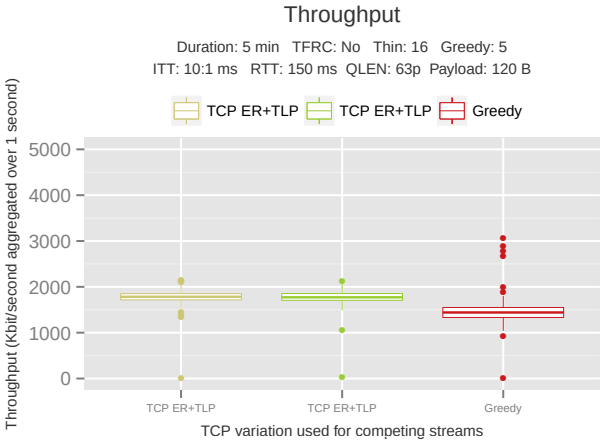
(c)

Figure A.2.40



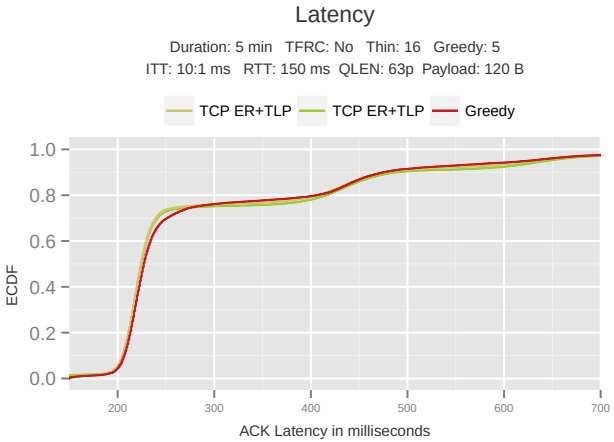
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	40 / 43 / 46	519/569/608	6/1/4/6/8/23
TCP ER+TLP	rdbsender	16	42 / 45 / 48	551/594/638	6/1/4/6/8/25
Greedy	zsender	5	37 / 39 / 43	1446/1446/1447	6/1/4/6/8/26

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
111 K	5681	5694	0	63.2 M	60.3 M	2.9 M	2.9 M	4.58 / 5.11	23492
106 K	5817	5817	0	62.8 M	59.6 M	3.2 M	3.2 M	5.13 / 5.49	22590
38.3 K	1972	1973	0	55.4 M	52.5 M	2.9 M	2.9 M	5.16 / 5.15	7504

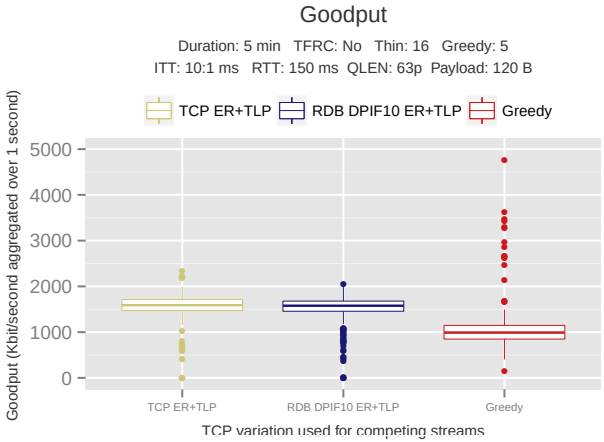
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
295	224	230	239	407	475	585	630	647	670	747	922	1718	3405	3548
300	225	230	241	416	489	612	644	662	693	850	955	1524	2190	2754
298	227	234	251	406	476	552	625	647	675	781	952	1895	3414	4249

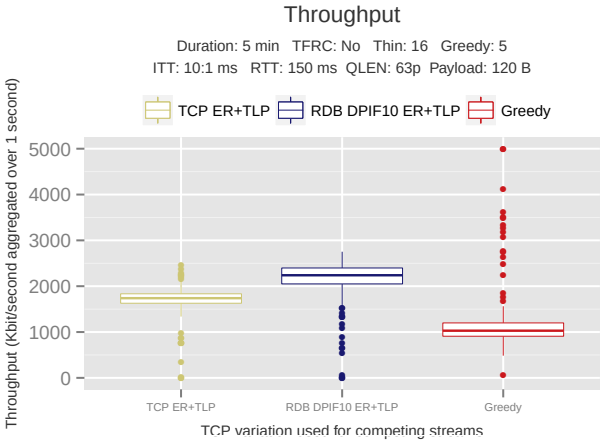
(c)

Figure A.2.41



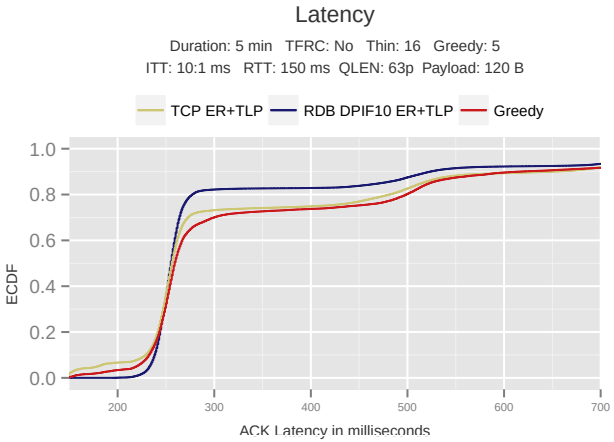
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	51 / 57 / 64	669/749/838	6/1/4/5/7/23
RDB DPIF10 ER+TLP	rdbsender	16	56 / 65 / 75	1090/1124/1159	6/1/4/5/7/19
Greedy	zsender	5	44 / 49 / 54	1445/1446/1447	7/1/4/5/8/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
84.2 K	5065	5076	0	62.9 M	59.2 M	3.6 M	3.6 M	5.74 / 6.01	17774
72.8 K	4891	3241	28.9 K	81.9 M	56.3 M	25.5 M	3.6 M	6.69 / 6.71	9718
30.7 K	1967	1967	0	44.4 M	41.5 M	2.8 M	2.8 M	6.42 / 6.41	6685

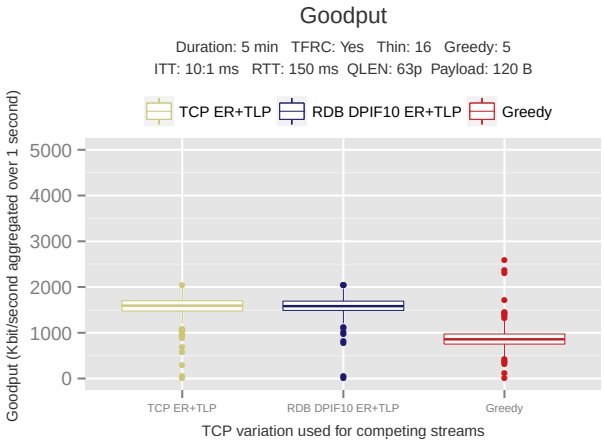
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
348	256	262	273	481	651	713	748	773	922	1006	1218	2058	7769	16540
336	255	259	264	277	524	691	740	768	945	1018	1252	2577	20059	36380
366	259	267	300	499	617	734	768	793	953	1015	1263	4799	7668	8363

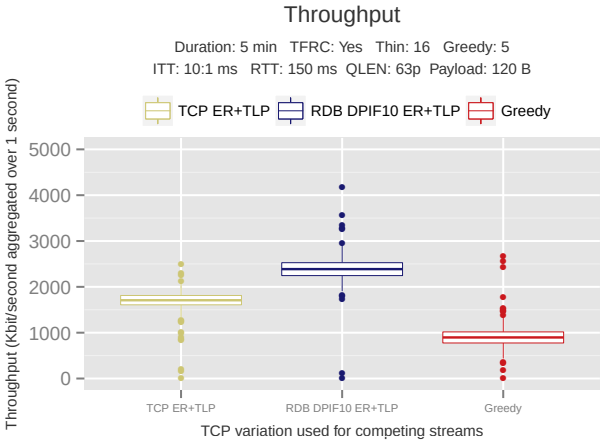
(c)

Figure A.2.42



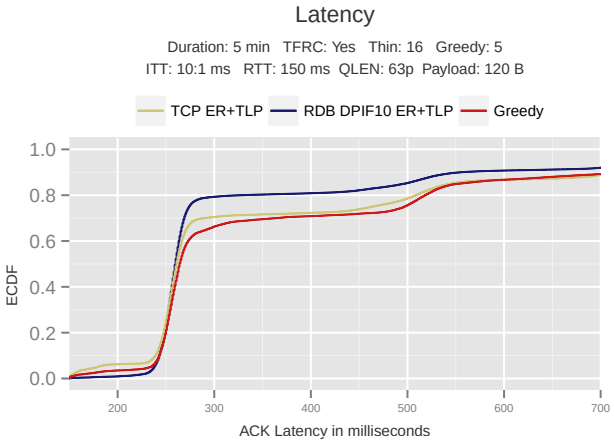
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	58 / 64 / 71	783/840/945	6/1/4/5/7/24
RDB DPIF10 ER+TLP	rdbsender	16	47 / 57 / 63	1037/1094/1137	7/1/4/6/8/128
Greedy	zsender	5	57 / 60 / 66	1444/1445/1447	5/1/4/5/6/21

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
75.7 K	5461	5474	0	63.5 M	59.1 M	4.3 M	4.3 M	6.80 / 7.21	16620
83.9 K	6914	4157	36.3 K	91.7 M	58.8 M	32.9 M	4.2 M	8.19 / 8.24	12702
24.9 K	1904	1906	0	36.0 M	33.2 M	2.8 M	2.8 M	7.65 / 7.65	5723

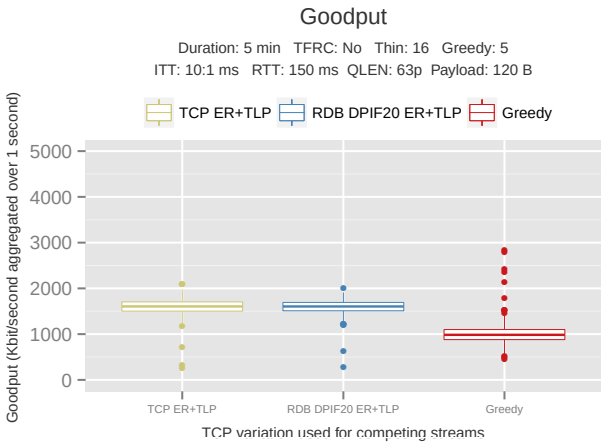
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
387	261	267	290	509	712	758	809	983	1031	1224	1471	4172	19024	28920
361	260	264	269	329	555	722	782	954	1028	1228	1507	4097	12621	12700
393	264	273	364	518	727	771	810	979	1034	1235	1468	4740	7817	7841

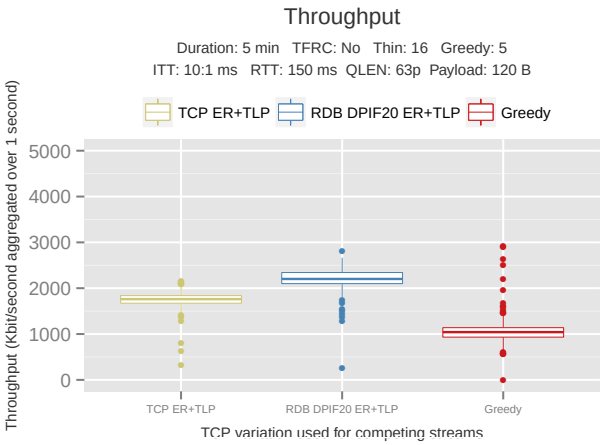
(c)

Figure A.2.43



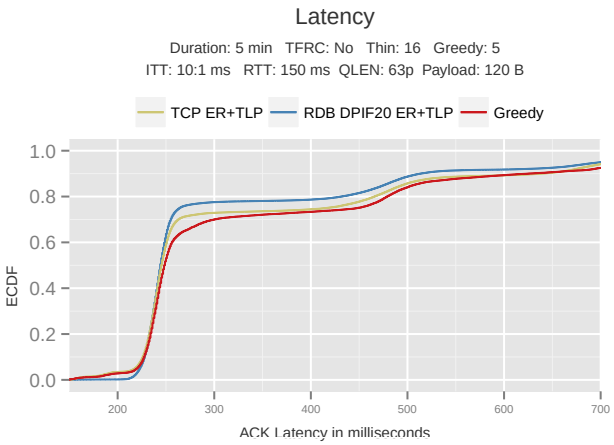
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	50 / 53 / 58	669/710/782	6/1/4/5/7/24
RDB DPIF20 ER+TLP	rdbsender	16	50 / 54 / 59	879/937/997	6/1/4/6/7/19
Greedy	zsender	5	49 / 53 / 58	1445/1446/1447	6/1/4/5/6/30

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
90.1 K	5674	5674	0	63.9 M	60.4 M	3.5 M	3.5 M	5.47 / 6.29	19448
87.8 K	5860	4560	24.0 K	82.1 M	59.5 M	22.5 M	3.7 M	6.43 / 6.67	15899
28.5 K	1936	1936	0	41.2 M	38.4 M	2.8 M	2.8 M	6.80 / 6.80	6327

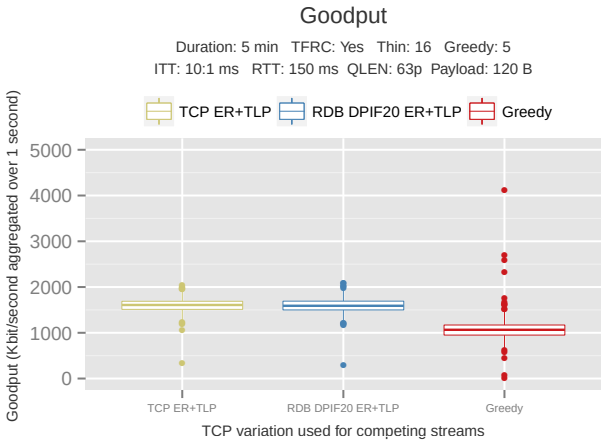
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
339	245	251	263	466	639	686	715	740	895	963	1177	2138	2759	5027
325	244	249	255	432	514	663	701	728	878	959	1169	2003	3089	4687
361	249	256	300	479	625	707	739	800	937	999	1227	5499	11148	12300

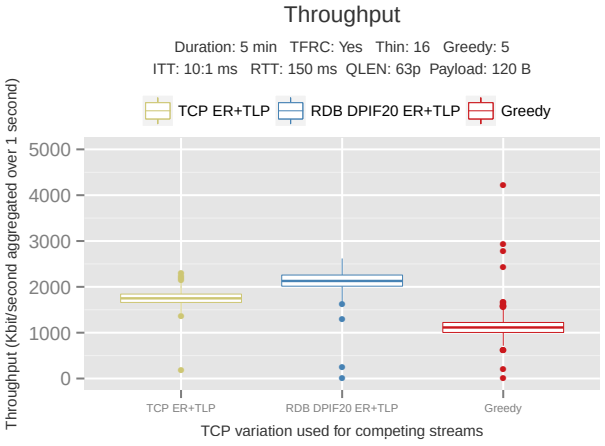
(c)

Figure A.2.44



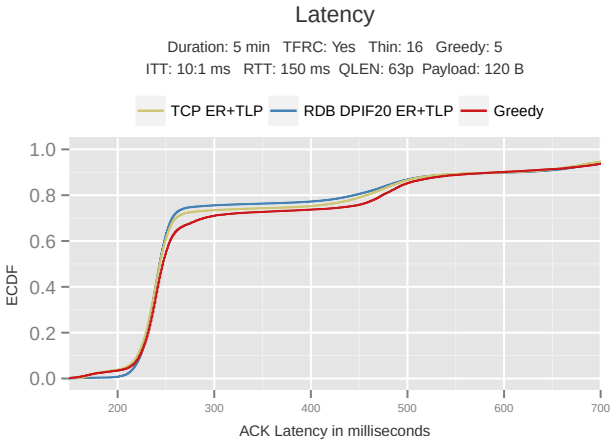
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	47 / 52 / 55	616/688/749	6/1/4/6/7/26
RDB DPIF20 ER+TLP	rdbsender	16	51 / 55 / 59	862/918/972	6/1/4/6/8/25
Greedy	zsender	5	46 / 49 / 51	1445/1446/1447	6/1/4/5/7/27

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
93.0 K	5649	5649	0	63.9 M	60.4 M	3.5 M	3.5 M	5.51 / 6.07	19997
86.7 K	6105	4902	20.4 K	79.6 M	59.5 M	20.1 M	4.0 M	6.83 / 7.04	16486
30.5 K	2055	2057	0	44.1 M	41.2 M	3.0 M	3.0 M	6.74 / 6.73	6621

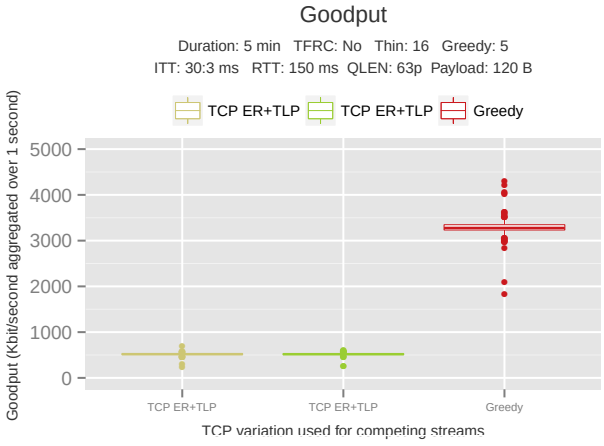
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
332	244	249	260	457	603	677	707	729	841	947	1160	1830	2760	3110
338	243	248	256	445	612	686	736	895	955	1132	1341	2455	4019	6131
346	247	254	289	474	594	689	723	747	872	963	1201	4045	4967	5517

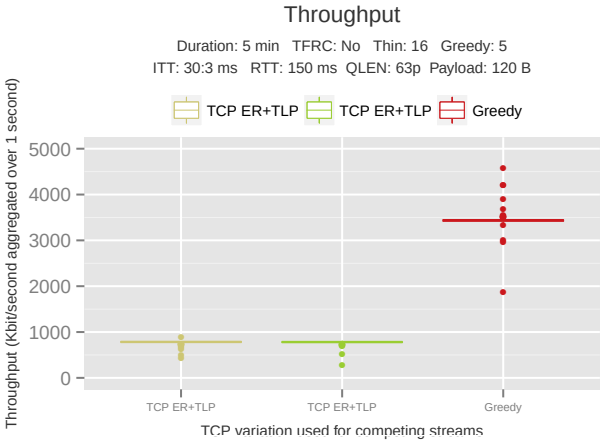
(c)

Figure A.2.45



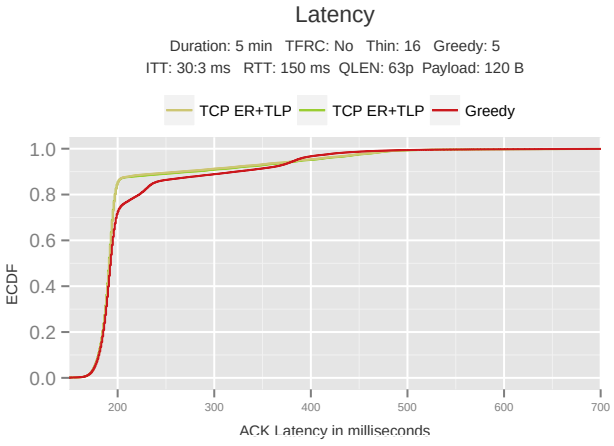
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	31 / 32 / 32	129/131/133	9/1/7/10/11/13
TCP ER+TLP	rdbsender	16	31 / 32 / 32	129/131/134	9/1/7/10/11/13
Greedy	zsender	5	16 / 17 / 18	1447/1447/1448	11/3/9/11/13/36

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
151 K	2467	2467	0	19.8 M	19.5 M	316 K	316 K	1.59 / 1.63	16039
149 K	2514	2514	0	19.6 M	19.3 M	321 K	321 K	1.64 / 1.68	16457
87.5 K	1633	1633	0	127 M	124 M	2.4 M	2.4 M	1.87 / 1.87	11690

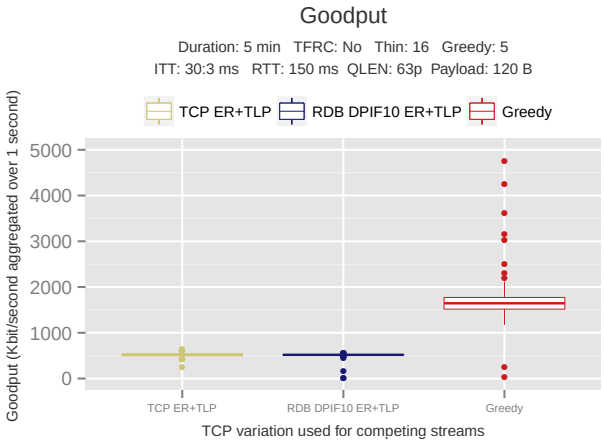
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
211	191	193	195	197	264	346	389	413	438	460	482	718	1617	2682
212	191	193	195	197	278	353	396	418	441	463	484	697	1045	2164
218	193	195	199	222	323	371	384	392	406	426	465	688	991	1059

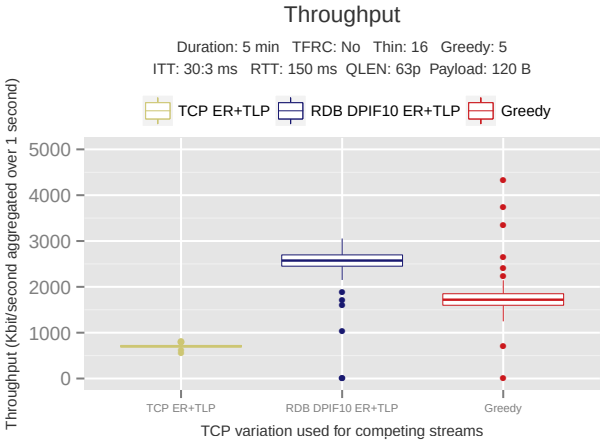
(c)

Figure A.2.46



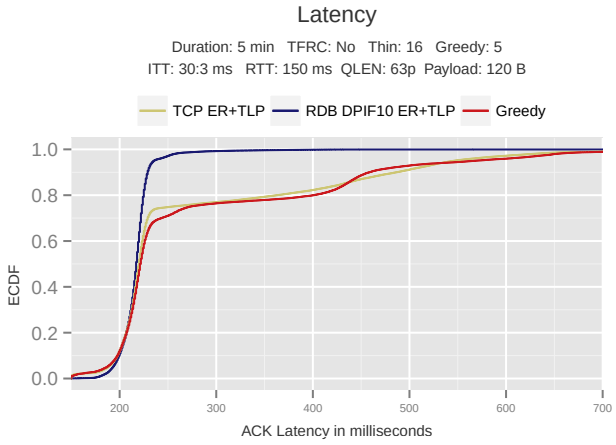
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	41 / 43 / 45	172/184/192	6/1/5/6/8/12
RDB DPIF10 ER+TLP	rdbsender	16	43 / 46 / 48	899/904/907	6/1/4/6/8/12
Greedy	zsender	5	32 / 32 / 34	1446/1447/1447	7/1/5/7/8/26

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
111 K	5040	5040	0	20.3 M	19.5 M	801 K	801 K	3.94 / 4.55	23253
103 K	4774	49	103 K	93.2 M	19.2 M	74.0 M	12.6 K	4.71 / 4.63	16
46.0 K	2326	2328	0	66.6 M	63.2 M	3.4 M	3.4 M	5.05 / 5.05	9256

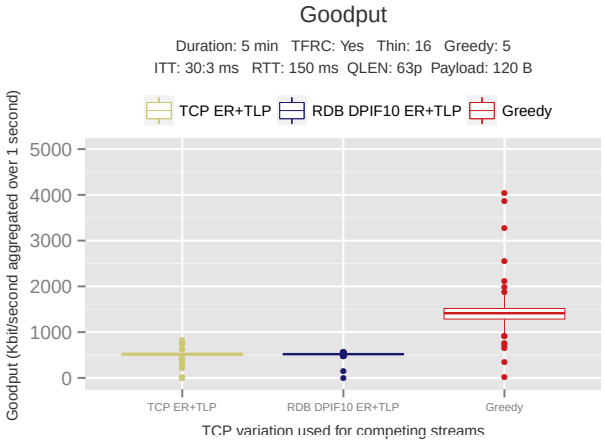
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
277	220	224	229	363	486	521	546	566	593	627	694	1119	1944	2562
218	217	220	222	225	229	232	235	242	249	257	284	420	1153	1671
280	222	226	243	401	459	501	567	602	630	652	706	1025	1309	1502

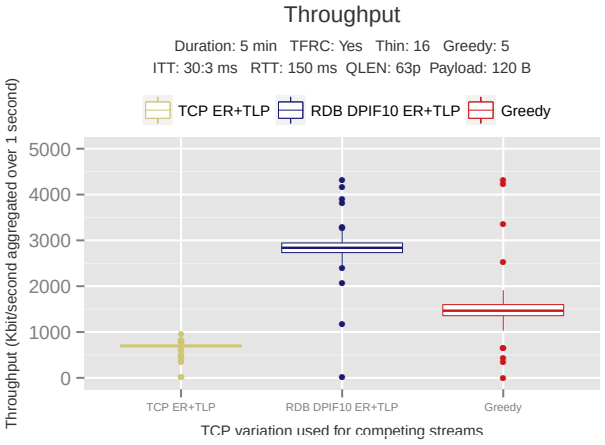
(c)

Figure A.2.47



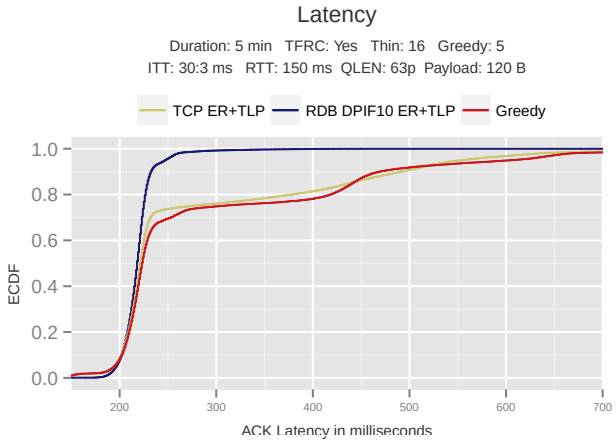
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	41 / 46 / 48	174/194/204	6/1/4/6/8/13
RDB DPIF10 ER+TLP	rdbsender	16	40 / 42 / 43	914/916/920	6/1/4/6/8/103
Greedy	zsender	5	37 / 38 / 41	1446/1447/1447	6/1/5/6/7/27

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
106 K	5077	5091	0	20.4 M	19.5 M	915 K	915 K	4.47 / 4.80	22961
113 K	5612	36	113 K	104 M	19.2 M	84.4 M	7.2 K	5.06 / 4.96	23
39.2 K	2198	2200	0	56.8 M	53.6 M	3.2 M	3.2 M	5.60 / 5.60	8274

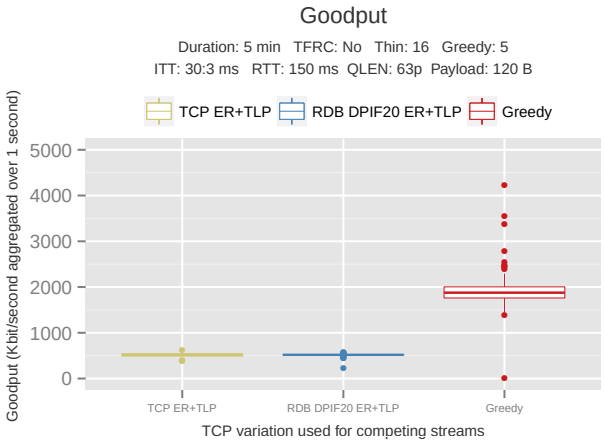
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
284	221	225	232	378	492	524	553	576	606	637	751	1477	4180	10890
220	218	221	223	227	233	238	247	251	255	261	287	409	1315	1800
299	223	229	253	418	469	533	606	631	648	669	856	3539	9954	9957

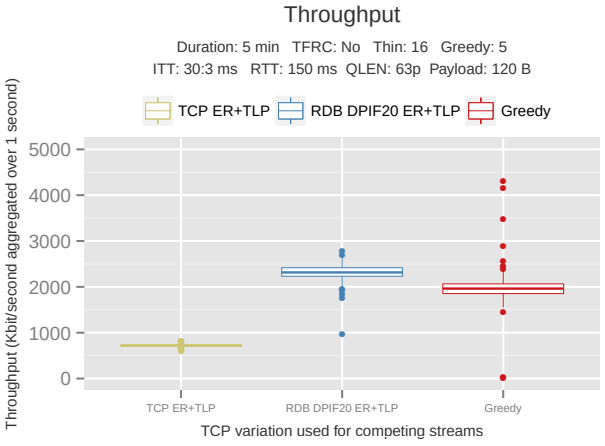
(c)

Figure A.2.48



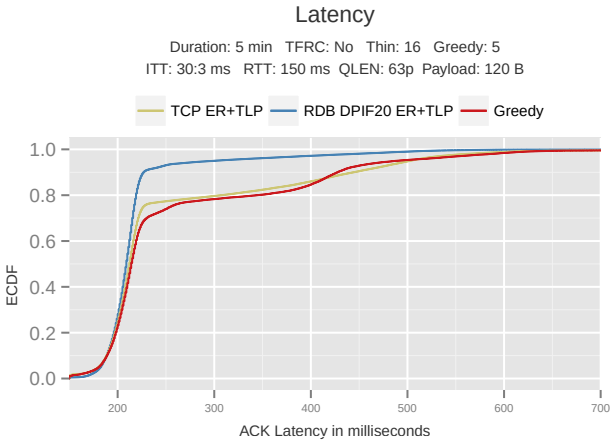
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	38 / 40 / 42	160/170/180	7/1/5/6/8/13
RDB DPIF20 ER+TLP	rdbsender	16	39 / 41 / 42	687/706/727	7/1/5/6/8/13
Greedy	zsender	5	27 / 29 / 31	1447/1447/1447	8/1/6/7/9/29

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
119 K	4788	4788	0	20.3 M	19.5 M	742 K	742 K	3.66 / 4.02	23860
116 K	4703	944	92.3 K	82.1 M	19.3 M	62.8 M	116 K	4.05 / 4.04	6151
51.7 K	2291	2293	0	74.9 M	71.6 M	3.3 M	3.3 M	4.43 / 4.43	10176

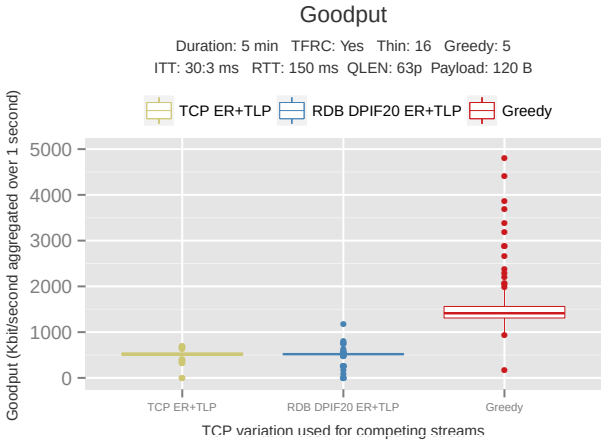
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
257	212	216	221	307	444	480	502	515	532	561	607	997	1502	2442
219	209	212	215	219	227	249	300	342	390	444	499	624	2238	2672
263	214	219	230	344	428	451	489	522	553	586	622	998	3919	4117

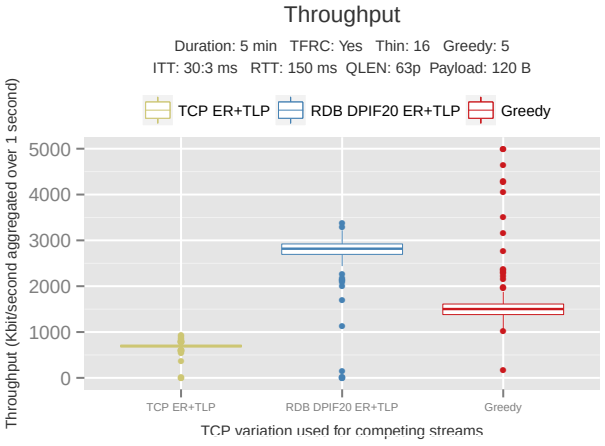
(c)

Figure A.2.49



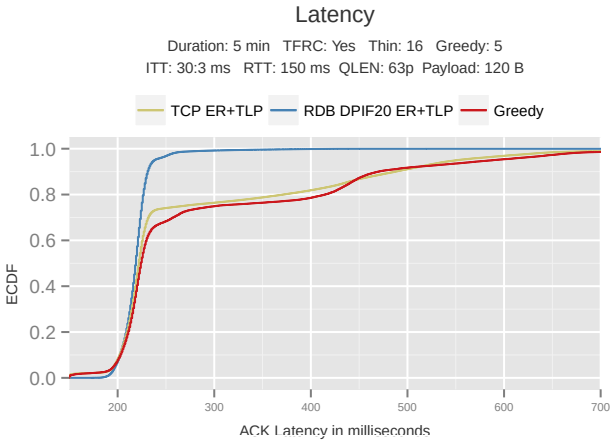
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	43 / 46 / 48	181/194/206	6/1/4/6/8/13
RDB DPIF20 ER+TLP	rdbsender	16	41 / 43 / 46	914/917/919	7/1/4/6/8/165
Greedy	zsender	5	34 / 36 / 38	1446/1446/1447	7/1/5/6/8/44

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
105 K	4838	4852	0	20.3 M	19.4 M	875 K	875 K	4.30 / 4.62	22560
110 K	5218	43	109 K	101 M	19.0 M	81.6 M	10.8 K	4.79 / 4.76	37
41.5 K	2147	2147	0	60.1 M	56.9 M	3.1 M	3.1 M	5.18 / 5.17	8771

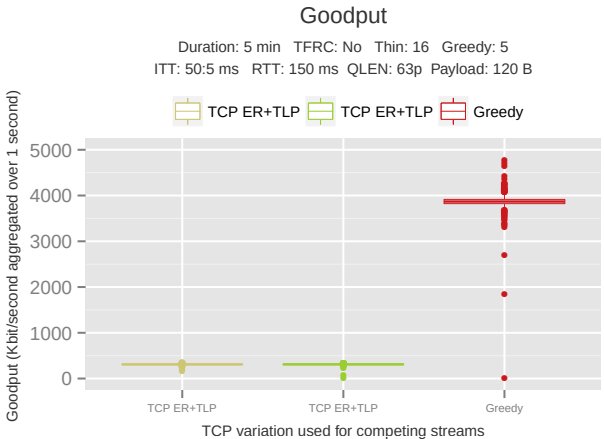
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
282	222	226	232	371	488	522	551	575	603	634	711	1246	4053	9038
221	219	221	224	227	231	234	238	245	251	257	284	417	3314	16540
294	224	230	258	415	470	535	589	617	643	668	830	2145	5713	7613

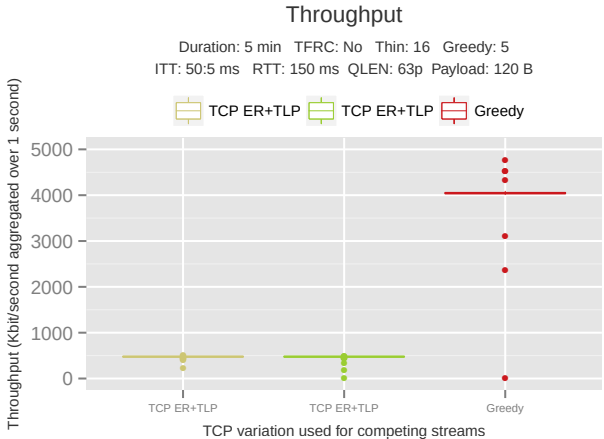
(c)

Figure A.2.50



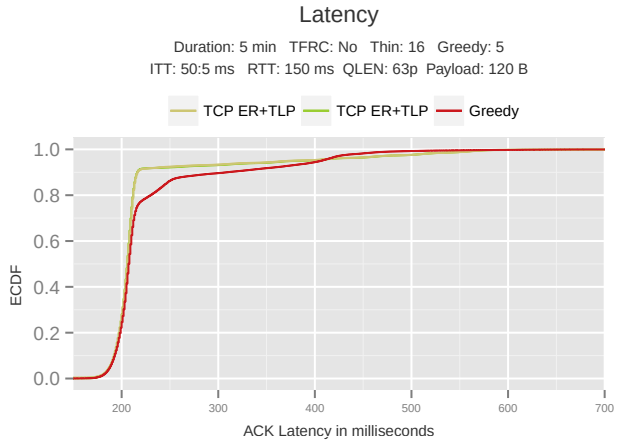
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	50 / 50 / 50	122/122/123	8/1/7/9/9/10
TCP ER+TLP	rdbsender	16	50 / 50 / 50	122/123/123	8/1/7/9/9/10
Greedy	zsender	5	14 / 14 / 15	1446/1447/1447	15/3/11/14/17/39

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
96.1 K	1245	1246	0	11.8 M	11.6 M	150 K	150 K	1.28 / 1.29	6845
94.8 K	1270	1283	0	11.6 M	11.5 M	155 K	155 K	1.33 / 1.34	6998
102 K	1328	1330	0	148 M	146 M	1.9 M	1.9 M	1.30 / 1.30	12512

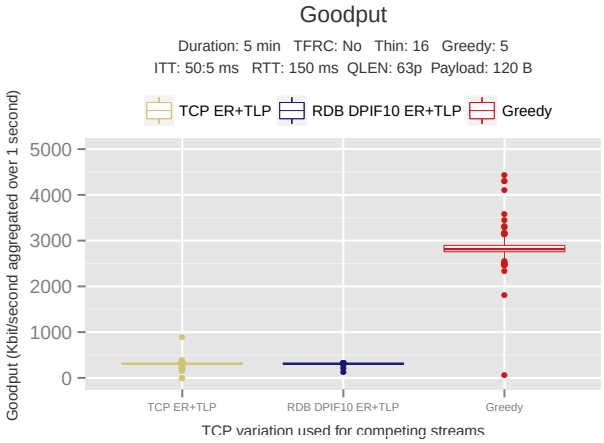
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
221	206	208	210	212	217	275	373	419	466	512	557	723	907	1121
222	206	208	210	212	216	293	382	421	467	512	556	679	1151	1153
230	207	210	213	230	309	376	406	415	423	443	476	644	818	1150

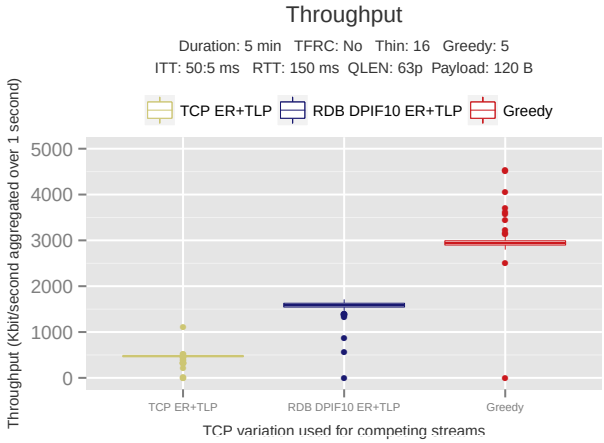
(c)

Figure A.2.51



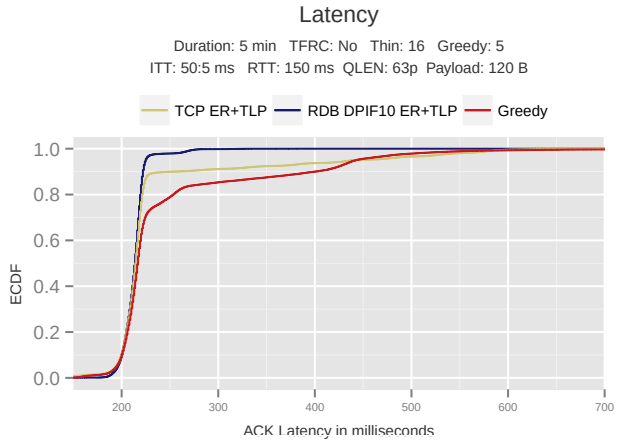
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	50 / 51 / 53	123/125/128	8/1/6/9/9/10
RDB DPIF10 ER+TLP	rdbsender	16	50 / 51 / 52	575/577/580	7/2/6/9/9/11
Greedy	zsender	5	18 / 20 / 22	1447/1447/1447	11/2/9/10/12/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
94.2 K	1585	1597	0	11.8 M	11.6 M	193 K	193 K	1.64 / 1.68	8680
92.9 K	1671	3	92.9 K	53.6 M	11.5 M	42.1 M	0	1.86 / 1.80	
75.4 K	1592	1592	0	109 M	107 M	2.3 M	2.3 M	2.11 / 2.11	11625

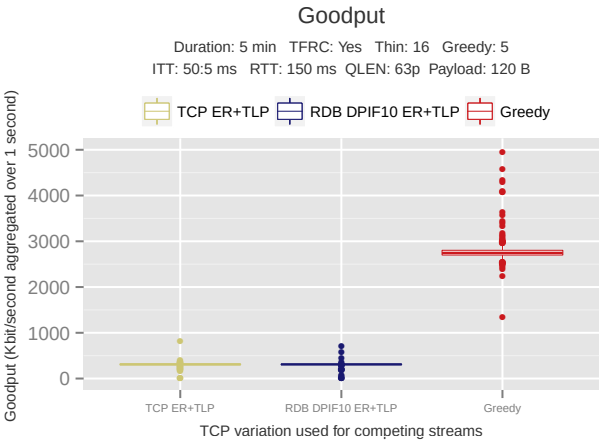
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
236	215	217	219	221	254	379	440	481	521	547	581	742	3245	7680
214	214	216	218	220	222	224	225	226	229	256	269	319	381	750
249	217	220	224	254	400	430	443	457	477	505	560	781	1006	1014

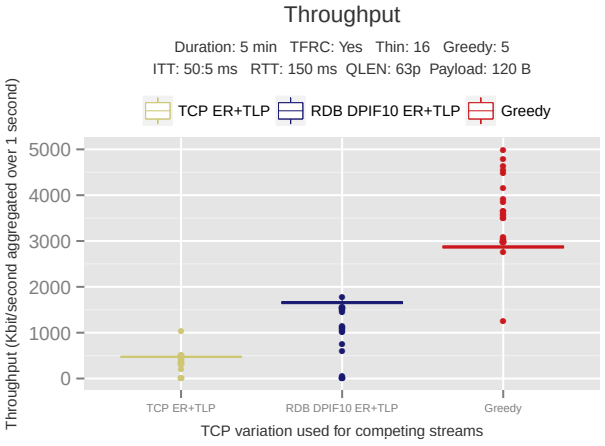
(c)

Figure A.2.52



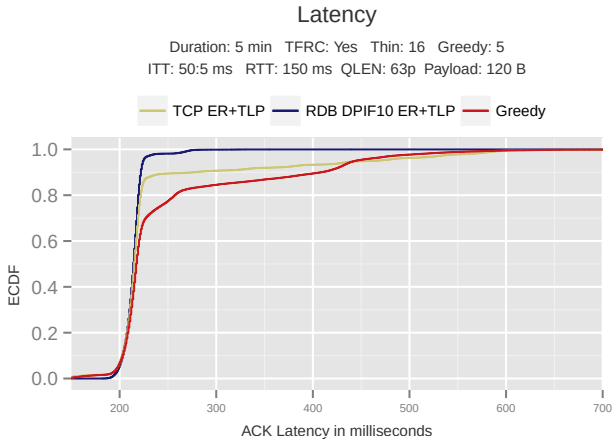
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	51 / 52 / 53	123/126/129	8/1/6/9/9/10
RDB DPIF10 ER+TLP	rdbsender	16	50 / 51 / 52	588/590/592	10/1/8/9/9/120
Greedy	zsender	5	19 / 20 / 21	1447/1447/1447	11/3/9/11/13/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
93.8 K	1663	1675	0	11.8 M	11.6 M	205 K	205 K	1.74 / 1.77	8971
93.2 K	1746	33	93.0 K	55.0 M	11.3 M	43.6 M	2.6 K	1.87 / 1.87	14
74.6 K	1571	1571	0	108 M	106 M	2.3 M	2.3 M	2.11 / 2.11	11916

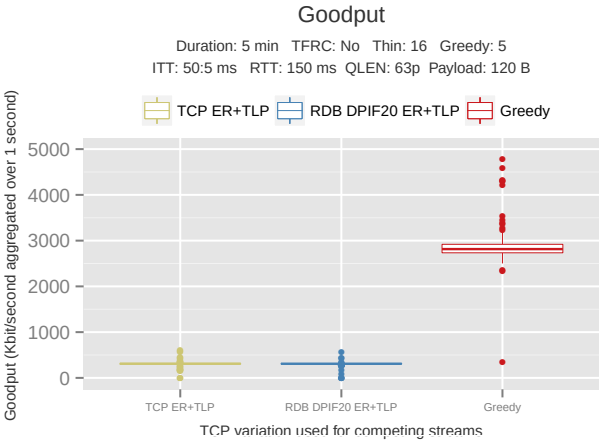
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
239	215	217	219	222	276	387	464	488	528	561	586	798	3260	7732
216	214	216	218	219	222	223	224	226	229	239	268	316	3755	10530
252	217	220	227	257	409	432	444	460	479	511	559	714	819	999

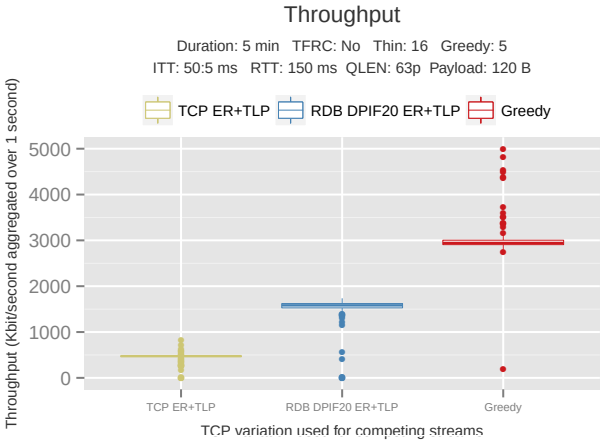
(c)

Figure A.2.53



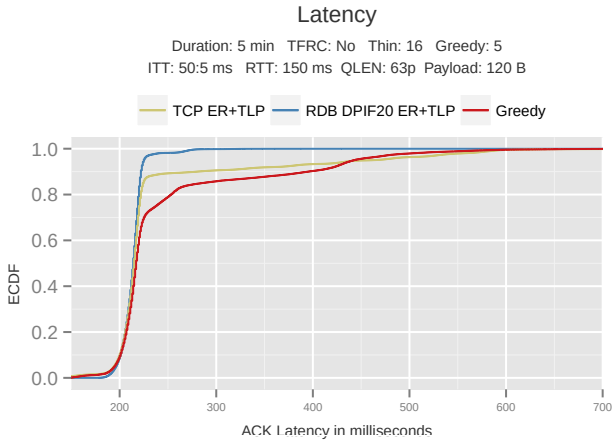
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	51 / 52 / 54	124/127/133	7/1/6/9/9/10
RDB DPIF20 ER+TLP	rdbsender	16	51 / 52 / 57	576/578/585	7/1/6/8/9/10
Greedy	zsender	5	18 / 19 / 20	1447/1447/1447	11/3/9/11/12/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
93.0 K	1664	1674	0	11.8 M	11.6 M	218 K	218 K	1.84 / 1.79	9039
90.8 K	1675	31	90.7 K	52.6 M	11.3 M	41.2 M	4.6 K	1.86 / 1.84	8
76.2 K	1540	1540	0	110 M	108 M	2.2 M	2.2 M	2.02 / 2.02	11278

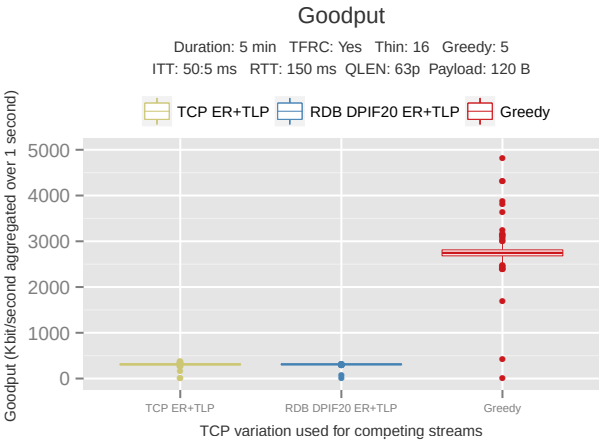
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
239	215	217	219	222	280	389	462	487	526	556	585	814	7511	16990
215	214	216	218	220	222	224	225	226	230	240	269	321	3530	18340
248	217	220	225	253	395	429	443	457	475	503	557	696	851	999

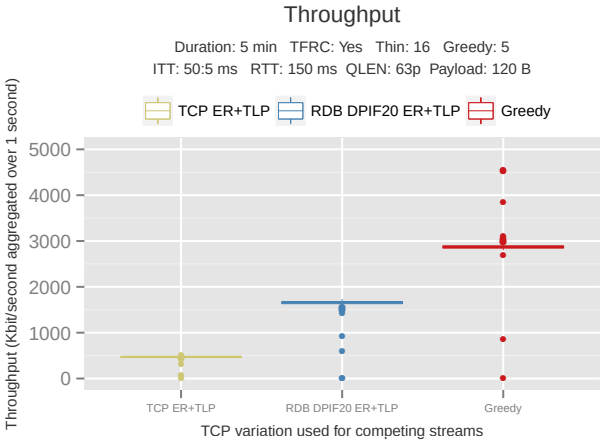
(c)

Figure A.2.54



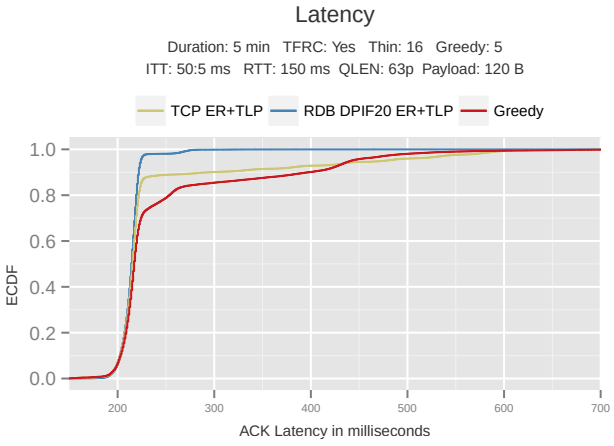
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	51 / 51 / 51	124/125/127	7/1/6/9/9/10
RDB DPIF20 ER+TLP	rdbsender	16	50 / 50 / 50	585/587/589	9/1/8/9/9/88
Greedy	zsender	5	19 / 20 / 22	1446/1447/1447	11/3/9/10/12/32

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
94.7 K	1808	1821	0	11.9 M	11.7 M	224 K	224 K	1.89 / 1.91	9444
95.4 K	1820	19	95.4 K	56.1 M	11.5 M	44.6 M	600	1.94 / 1.91	6
73.7 K	1557	1559	0	107 M	104 M	2.3 M	2.3 M	2.11 / 2.11	10957

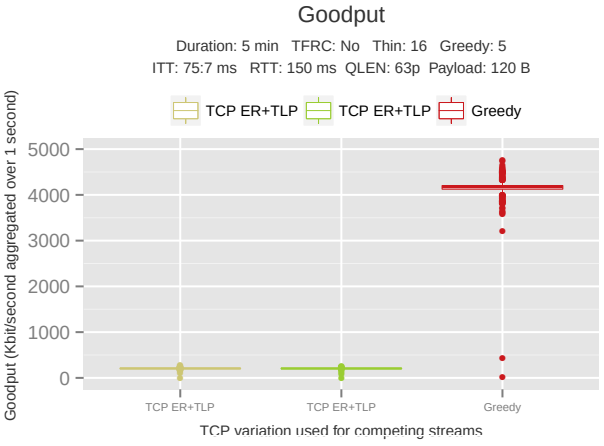
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
240	216	217	219	222	295	413	476	501	534	569	591	868	1151	1306
215	214	216	218	219	222	223	224	225	226	239	268	316	1151	1152
250	217	220	224	254	398	429	441	454	474	498	549	743	933	1153

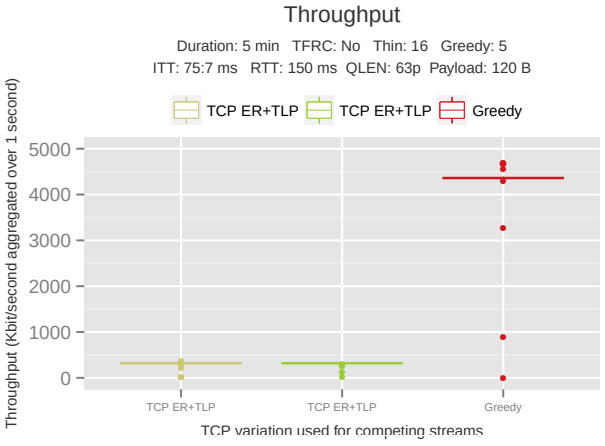
(c)

Figure A.2.55



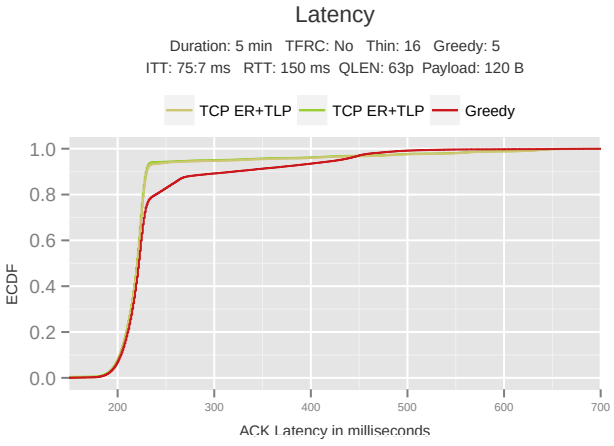
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 74 / 74	120/120/120	7/1/7/8/8/10
TCP ER+TLP	rdbsender	16	74 / 74 / 74	120/120/120	7/1/7/8/8/10
Greedy	zsender	5	13 / 13 / 14	1447/1447/1447	17/4/13/16/19/40

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
65.2 K	746	760	0	7.8 M	7.8 M	89.6 K	89.6 K	1.14 / 1.14	3343
64.4 K	708	709	0	7.7 M	7.7 M	84.5 K	84.5 K	1.09 / 1.10	3126
110 K	1242	1244	0	159 M	157 M	1.8 M	1.8 M	1.13 / 1.13	13287

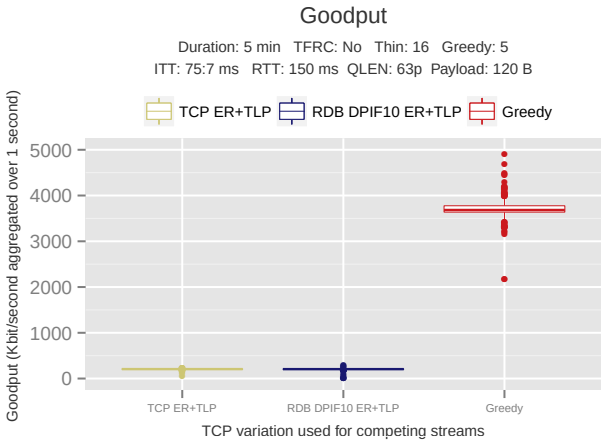
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
233	220	223	225	227	230	234	326	402	476	549	620	691	1151	1152
231	220	222	224	226	230	232	300	384	462	542	617	699	908	971
243	222	225	228	240	320	390	428	441	450	464	492	660	773	1151

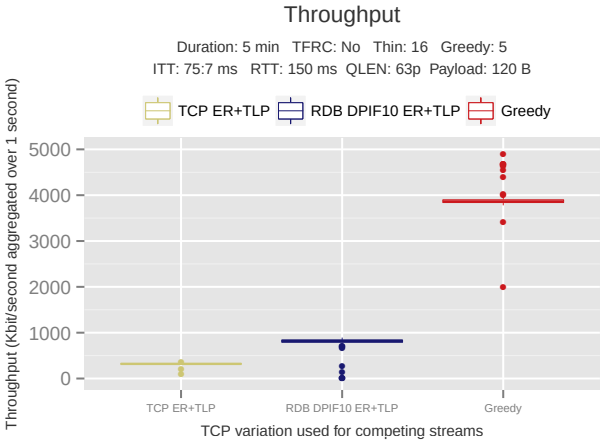
(c)

Figure A.2.56



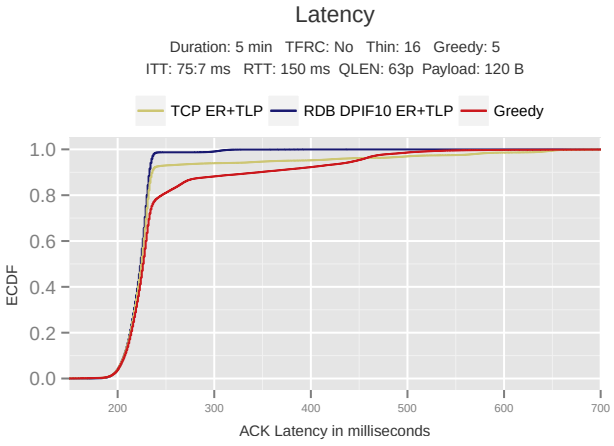
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 74 / 74	120/120/121	7/1/7/8/8/10
RDB DPIF10 ER+TLP	rdbsender	16	76 / 76 / 79	413/417/420	7/1/7/8/8/10
Greedy	zsender	5	15 / 15 / 16	1447/1447/1448	15/5/12/14/17/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
65.0 K	857	861	0	7.8 M	7.7 M	103 K	103 K	1.31 / 1.32	3737
62.4 K	799	30	62.3 K	26.0 M	7.5 M	18.5 M	1.9 K	1.34 / 1.28	11
98.1 K	1277	1277	0	142 M	140 M	1.8 M	1.8 M	1.30 / 1.30	12250

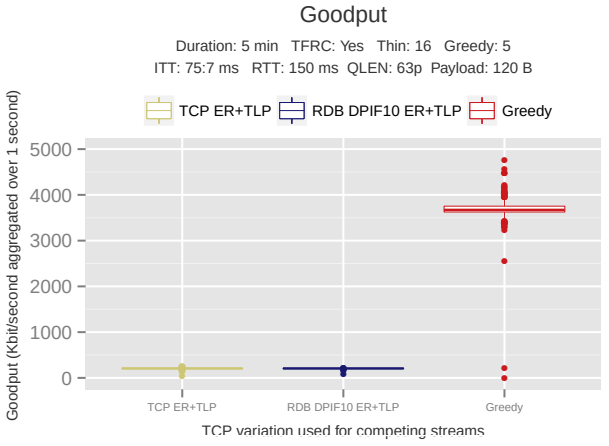
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
239	225	227	230	232	236	250	371	437	502	570	640	701	934	1152
224	224	226	228	230	233	234	235	236	236	238	298	328	3253	10550
249	226	229	233	246	346	415	447	456	464	482	514	683	768	999

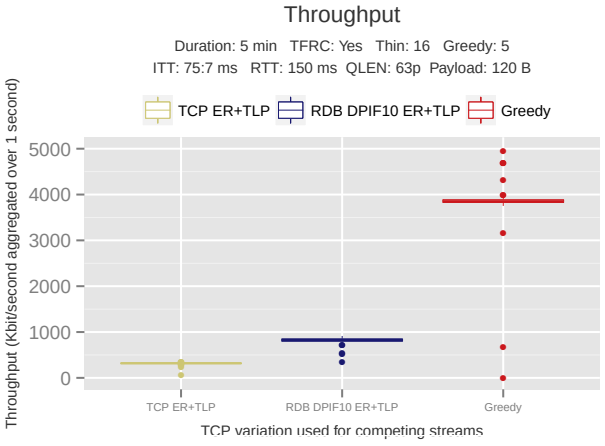
(c)

Figure A.2.57



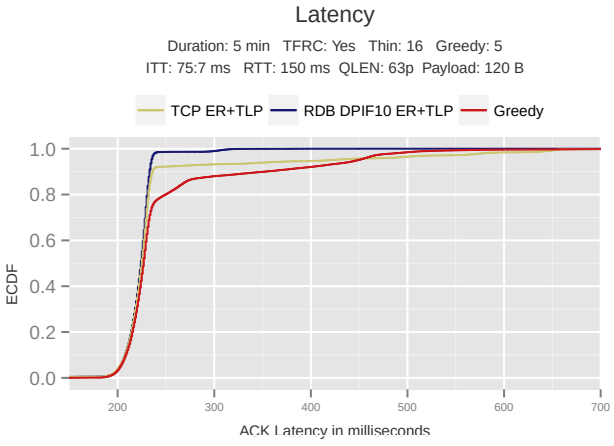
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 74 / 74	120/121/121	7/1/7/8/8/10
RDB DPIF10 ER+TLP	rdbsender	16	74 / 75 / 75	418/419/421	8/3/8/8/9/22
Greedy	zsender	5	14 / 15 / 16	1446/1447/1447	15/3/12/14/17/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
65.2 K	984	989	0	7.9 M	7.7 M	119 K	119 K	1.51 / 1.51	4219
63.7 K	883	9	63.7 K	26.7 M	7.6 M	19.1 M	1.1 K	1.48 / 1.38	6
97.6 K	1321	1323	0	141 M	139 M	1.9 M	1.9 M	1.35 / 1.35	12466

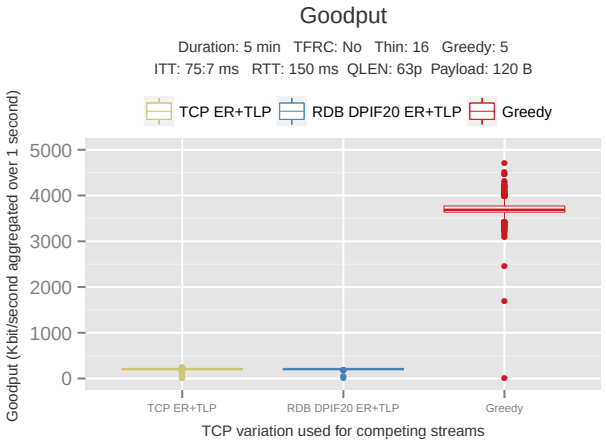
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
242	225	227	230	232	236	289	419	482	523	578	644	760	942	1154
223	224	227	229	231	233	234	235	236	237	238	301	327	449	680
251	227	230	233	250	353	418	449	457	465	487	523	744	1102	1152

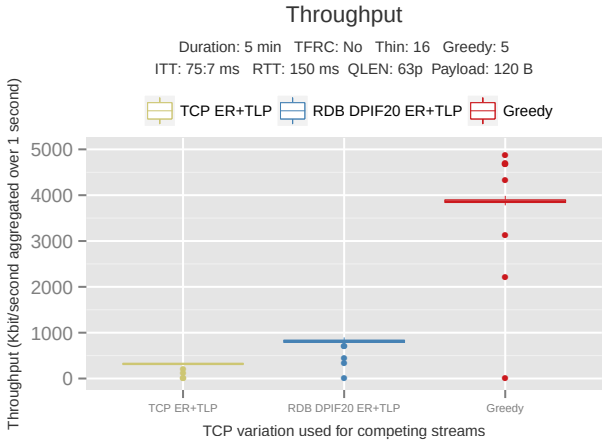
(c)

Figure A.2.58



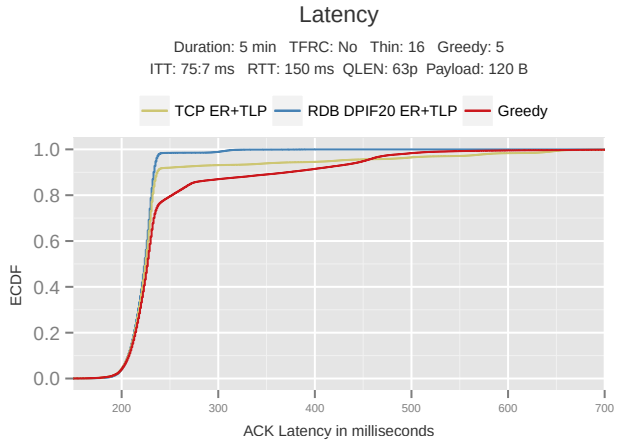
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 74 / 74	120/121/121	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	16	75 / 75 / 76	413/416/418	7/2/7/8/8/10
Greedy	zsender	5	15 / 15 / 16	1446/1447/1447	15/4/12/15/18/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
65.0 K	1007	1022	0	7.8 M	7.7 M	122 K	122 K	1.55 / 1.55	4191
63.2 K	904	15	63.2 K	26.3 M	7.6 M	18.7 M	240	1.53 / 1.43	2
98.1 K	1403	1405	0	142 M	140 M	2.0 M	2.0 M	1.43 / 1.43	13393

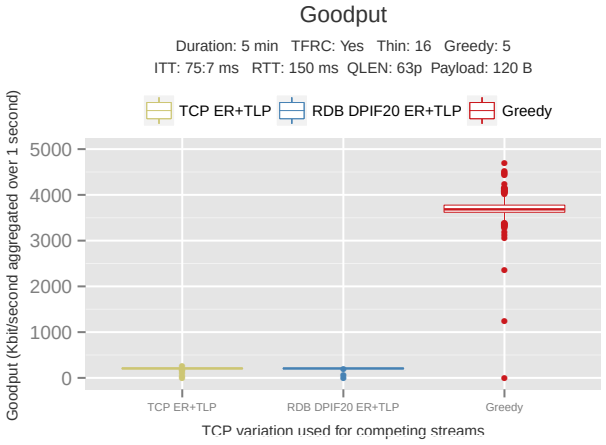
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
242	225	227	230	232	237	291	420	484	531	579	643	760	1152	1153
223	224	226	229	231	233	235	236	236	237	239	301	366	1152	1153
253	227	230	233	252	371	427	452	460	469	491	527	729	943	1150

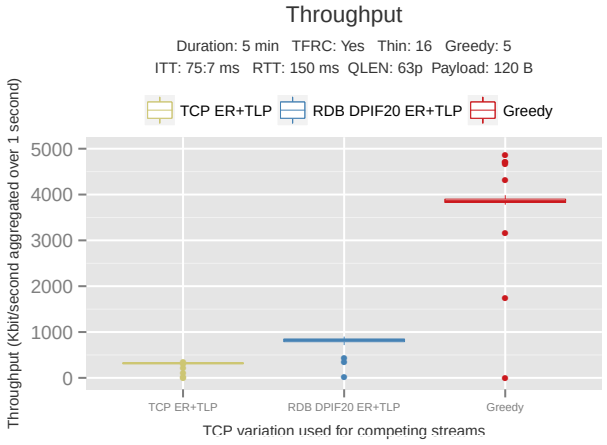
(c)

Figure A.2.59



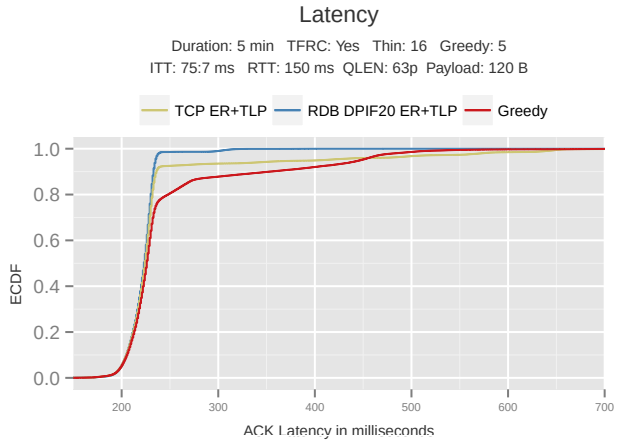
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 74 / 74	120/120/121	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	16	74 / 75 / 75	414/417/419	11/3/8/8/10/104
Greedy	zsender	5	15 / 15 / 16	1446/1447/1448	15/3/12/14/17/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
65.0 K	932	947	0	7.8 M	7.7 M	112 K	112 K	1.43 / 1.43	4014
63.5 K	864	16	63.5 K	26.5 M	7.6 M	18.9 M	360	1.45 / 1.36	3
97.8 K	1335	1337	0	142 M	140 M	1.9 M	1.9 M	1.36 / 1.36	12624

(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
240	224	227	229	232	236	271	407	461	510	573	639	740	1151	1241
222	223	226	228	230	233	234	235	236	237	239	300	330	1152	1153
250	226	229	233	248	353	420	447	456	464	483	514	702	1020	1150

(c)

Figure A.2.60

A.3 Latency tests with greedy cross traffic bundle limitation

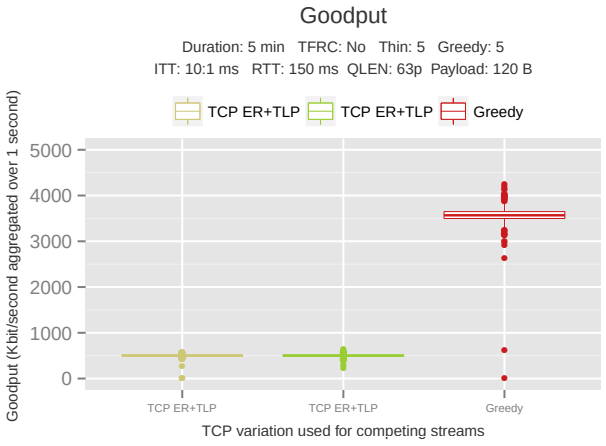
Table A.3

	Type	Streams	Cong	TFRC	RTT	ITT	Payload
Figure A.3.1	TCP ER+TLP	5	Rdb	No	150	10:1	120
Figure A.3.2	RDB DPIF10 ER+TLP	5	Rdb	No	150	10:1	120
Figure A.3.3	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	10:1	120
Figure A.3.4	RDB DPIF20 ER+TLP	5	Rdb	No	150	10:1	120
Figure A.3.5	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	10:1	120
Figure A.3.6	TCP ER+TLP	5	Rdb	No	150	30:3	120
Figure A.3.7	RDB DPIF10 ER+TLP	5	Rdb	No	150	30:3	120
Figure A.3.8	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	30:3	120
Figure A.3.9	RDB DPIF20 ER+TLP	5	Rdb	No	150	30:3	120
Figure A.3.10	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	30:3	120
Figure A.3.11	TCP ER+TLP	5	Rdb	No	150	50:5	120
Figure A.3.12	RDB DPIF10 ER+TLP	5	Rdb	No	150	50:5	120
Figure A.3.13	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	50:5	120
Figure A.3.14	RDB DPIF20 ER+TLP	5	Rdb	No	150	50:5	120
Figure A.3.15	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	50:5	120
Figure A.3.16	TCP ER+TLP	5	Rdb	No	150	75:7	120
Figure A.3.17	RDB DPIF10 ER+TLP	5	Rdb	No	150	75:7	120
Figure A.3.18	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	75:7	120
Figure A.3.19	RDB DPIF20 ER+TLP	5	Rdb	No	150	75:7	120
Figure A.3.20	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	75:7	120
Figure A.3.21	TCP ER+TLP	5	Rdb	No	150	100:10	120
Figure A.3.22	RDB DPIF10 ER+TLP	5	Rdb	No	150	100:10	120
Figure A.3.23	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	100:10	120
Figure A.3.24	RDB DPIF20 ER+TLP	5	Rdb	No	150	100:10	120
Figure A.3.25	RDB DPIF20 ER+TLP	5	Rdb	Yes	150	100:10	120
Figure A.3.26	TCP ER+TLP	10	Rdb	No	150	10:1	120
Figure A.3.27	RDB DPIF10 ER+TLP	10	Rdb	No	150	10:1	120
Figure A.3.28	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	10:1	120
Figure A.3.29	RDB DPIF20 ER+TLP	10	Rdb	No	150	10:1	120
Figure A.3.30	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	10:1	120
Figure A.3.31	TCP ER+TLP	10	Rdb	No	150	30:3	120
Figure A.3.32	RDB DPIF10 ER+TLP	10	Rdb	No	150	30:3	120
Figure A.3.33	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	30:3	120
Figure A.3.34	RDB DPIF20 ER+TLP	10	Rdb	No	150	30:3	120
Figure A.3.35	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	30:3	120
Figure A.3.36	TCP ER+TLP	10	Rdb	No	150	50:5	120
Figure A.3.37	RDB DPIF10 ER+TLP	10	Rdb	No	150	50:5	120
Figure A.3.38	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	50:5	120
Figure A.3.39	RDB DPIF20 ER+TLP	10	Rdb	No	150	50:5	120
Figure A.3.40	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	50:5	120
Figure A.3.41	TCP ER+TLP	10	Rdb	No	150	75:7	120
Figure A.3.42	RDB DPIF10 ER+TLP	10	Rdb	No	150	75:7	120
Figure A.3.43	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	75:7	120
Figure A.3.44	RDB DPIF20 ER+TLP	10	Rdb	No	150	75:7	120

Figure A.3.45	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	75:7	120
Figure A.3.46	TCP ER+TLP	10	Rdb	No	150	100:10	120
Figure A.3.47	RDB DPIF10 ER+TLP	10	Rdb	No	150	100:10	120
Figure A.3.48	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	100:10	120
Figure A.3.49	RDB DPIF20 ER+TLP	10	Rdb	No	150	100:10	120
Figure A.3.50	RDB DPIF20 ER+TLP	10	Rdb	Yes	150	100:10	120
Figure A.3.51	TCP ER+TLP	13	Rdb	No	150	10:1	120
Figure A.3.52	RDB DPIF10 ER+TLP	13	Rdb	No	150	10:1	120
Figure A.3.53	RDB DPIF10 ER+TLP	13	Rdb	Yes	150	10:1	120
Figure A.3.54	RDB DPIF20 ER+TLP	13	Rdb	No	150	10:1	120
Figure A.3.55	RDB DPIF20 ER+TLP	13	Rdb	Yes	150	10:1	120
Figure A.3.56	TCP ER+TLP	13	Rdb	No	150	30:3	120
Figure A.3.57	RDB DPIF10 ER+TLP	13	Rdb	No	150	30:3	120
Figure A.3.58	RDB DPIF10 ER+TLP	13	Rdb	Yes	150	30:3	120
Figure A.3.59	RDB DPIF20 ER+TLP	13	Rdb	No	150	30:3	120
Figure A.3.60	RDB DPIF20 ER+TLP	13	Rdb	Yes	150	30:3	120
Figure A.3.61	TCP ER+TLP	13	Rdb	No	150	50:5	120
Figure A.3.62	RDB DPIF10 ER+TLP	13	Rdb	No	150	50:5	120
Figure A.3.63	RDB DPIF10 ER+TLP	13	Rdb	Yes	150	50:5	120
Figure A.3.64	RDB DPIF20 ER+TLP	13	Rdb	No	150	50:5	120
Figure A.3.65	RDB DPIF20 ER+TLP	13	Rdb	Yes	150	50:5	120
Figure A.3.66	TCP ER+TLP	13	Rdb	No	150	75:7	120
Figure A.3.67	RDB DPIF10 ER+TLP	13	Rdb	No	150	75:7	120
Figure A.3.68	RDB DPIF10 ER+TLP	13	Rdb	Yes	150	75:7	120
Figure A.3.69	RDB DPIF20 ER+TLP	13	Rdb	No	150	75:7	120
Figure A.3.70	RDB DPIF20 ER+TLP	13	Rdb	Yes	150	75:7	120
Figure A.3.71	TCP ER+TLP	13	Rdb	No	150	100:10	120
Figure A.3.72	RDB DPIF10 ER+TLP	13	Rdb	No	150	100:10	120
Figure A.3.73	RDB DPIF10 ER+TLP	13	Rdb	Yes	150	100:10	120
Figure A.3.74	RDB DPIF20 ER+TLP	13	Rdb	No	150	100:10	120
Figure A.3.75	RDB DPIF20 ER+TLP	13	Rdb	Yes	150	100:10	120
Figure A.3.76	TCP ER+TLP	16	Rdb	No	150	10:1	120
Figure A.3.77	RDB DPIF10 ER+TLP	16	Rdb	No	150	10:1	120
Figure A.3.78	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	10:1	120
Figure A.3.79	RDB DPIF20 ER+TLP	16	Rdb	No	150	10:1	120
Figure A.3.80	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	10:1	120
Figure A.3.81	TCP ER+TLP	16	Rdb	No	150	30:3	120
Figure A.3.82	RDB DPIF10 ER+TLP	16	Rdb	No	150	30:3	120
Figure A.3.83	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	30:3	120
Figure A.3.84	RDB DPIF20 ER+TLP	16	Rdb	No	150	30:3	120
Figure A.3.85	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	30:3	120
Figure A.3.86	TCP ER+TLP	16	Rdb	No	150	50:5	120
Figure A.3.87	RDB DPIF10 ER+TLP	16	Rdb	No	150	50:5	120
Figure A.3.88	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	50:5	120
Figure A.3.89	RDB DPIF20 ER+TLP	16	Rdb	No	150	50:5	120
Figure A.3.90	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	50:5	120
Figure A.3.91	TCP ER+TLP	16	Rdb	No	150	75:7	120
Figure A.3.92	RDB DPIF10 ER+TLP	16	Rdb	No	150	75:7	120
Figure A.3.93	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	75:7	120
Figure A.3.94	RDB DPIF20 ER+TLP	16	Rdb	No	150	75:7	120
Figure A.3.95	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	75:7	120
Figure A.3.96	TCP ER+TLP	16	Rdb	No	150	100:10	120

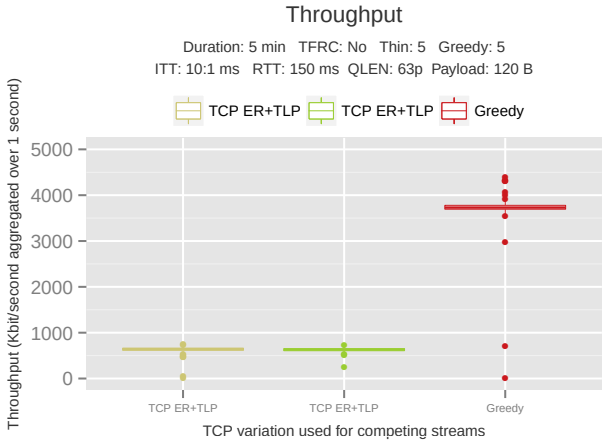
Figure A.3.97	RDB DPIF10 ER+TLP	16	Rdb	No	150	100:10	120
Figure A.3.98	RDB DPIF10 ER+TLP	16	Rdb	Yes	150	100:10	120
Figure A.3.99	RDB DPIF20 ER+TLP	16	Rdb	No	150	100:10	120
Figure A.3.100	RDB DPIF20 ER+TLP	16	Rdb	Yes	150	100:10	120

Table A.3: Test setup for experiment 3



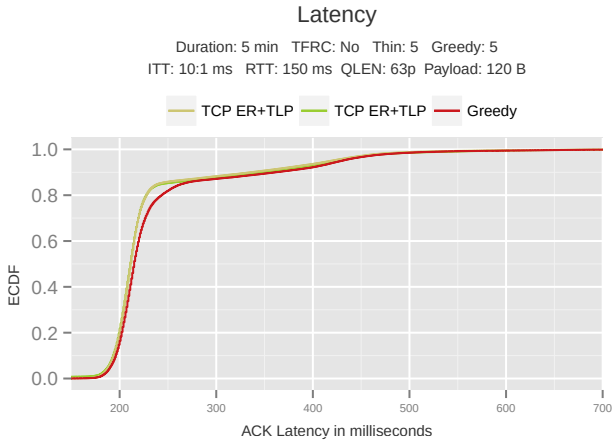
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	18 / 19 / 19	233/238/248	13/1/10/13/16/28
TCP ER+TLP	rdbsender	5	18 / 20 / 22	231/256/284	13/1/9/12/16/28
Greedy	zsender	5	14 / 16 / 17	1446/1447/1447	14/3/10/13/17/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
79.9 K	1109	1111	0	19.0 M	18.8 M	227.9 K	227.9 K	1.20 / 1.39	11240
74.0 K	1162	1162	0	18.9 M	18.6 M	235.1 K	235.1 K	1.25 / 1.57	10788
94.7 K	1649	1651	0	137.0 M	134.6 M	2.4 M	2.4 M	1.74 / 1.74	13121

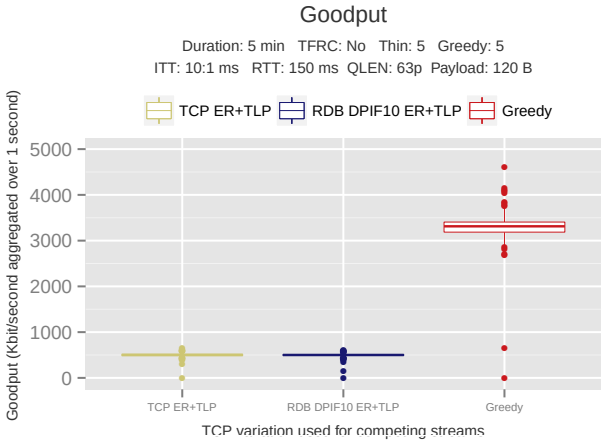
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
236	211	215	220	228	336	390	419	432	447	467	514	721	3347	3420
236	211	215	219	228	350	400	425	438	452	475	542	746	899	999
242	215	220	227	243	361	410	430	441	456	478	530	718	1254	1259

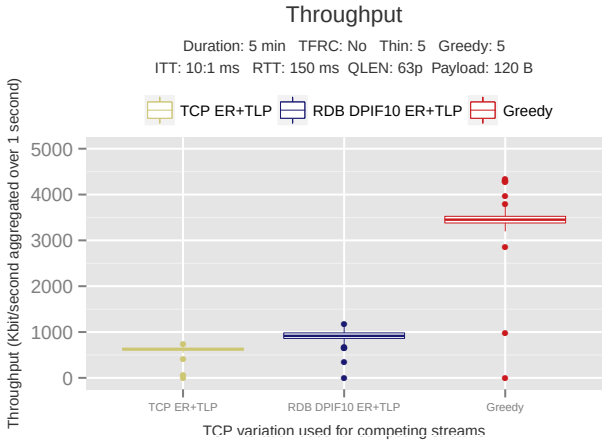
(c)

Figure A.3.1



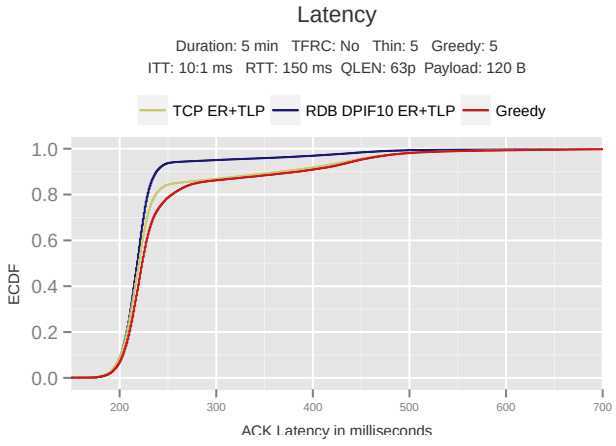
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	19 / 21 / 22	250/265/283	12/1/9/12/15/29
RDB DPIF10 ER+TLP	rdbsender	5	22 / 23 / 24	450/472/498	12/1/8/11/15/30
Greedy	zsender	5	15 / 17 / 19	1447/1447/1447	14/1/10/13/16/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
72.3 K	1157	1159	0	19.1 M	18.9 M	236.6 K	236.6 K	1.24 / 1.60	10635
64.3 K	1120	399	47.5 K	30.3 M	18.6 M	11.7 M	143.2 K	1.60 / 1.74	3565
87.9 K	1619	1620	0	127.3 M	124.9 M	2.3 M	2.3 M	1.84 / 1.84	12624

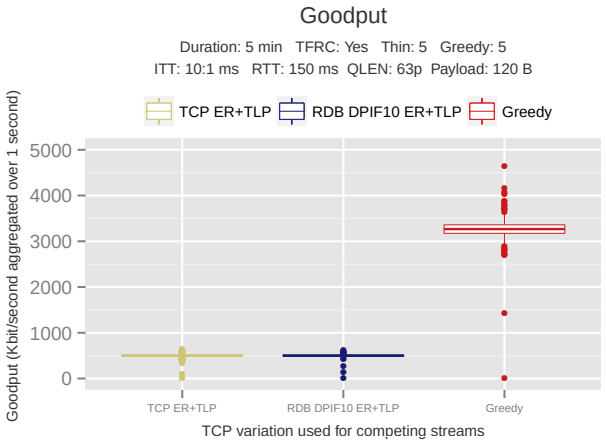
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	220	224	228	237	367	418	443	456	472	497	557	718	905	1152
229	218	221	225	229	238	246	298	355	402	438	476	773	1019	1707
253	223	228	235	254	385	428	447	460	474	496	554	843	2287	2299

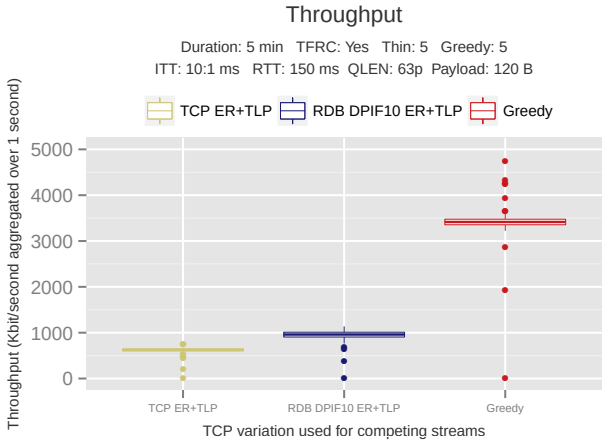
(c)

Figure A.3.2



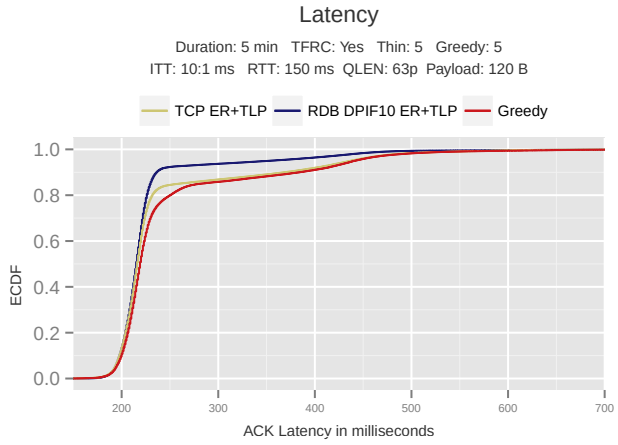
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	19 / 21 / 22	247/269/279	12/1/9/12/15/28
RDB DPIF10 ER+TLP	rdbsender	5	17 / 18 / 19	363/381/404	14/1/10/13/18/71
Greedy	zsender	5	16 / 17 / 18	1446/1447/1447	14/3/10/12/16/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
71.1 K	1188	1190	0	19.1 M	18.8 M	262.3 K	262.3 K	1.37 / 1.67	10563
80.7 K	1328	535	54.7 K	30.7 M	18.5 M	12.2 M	122.2 K	1.46 / 1.65	5897
87.1 K	1644	1646	0	126.1 M	123.7 M	2.4 M	2.4 M	1.89 / 1.89	12783

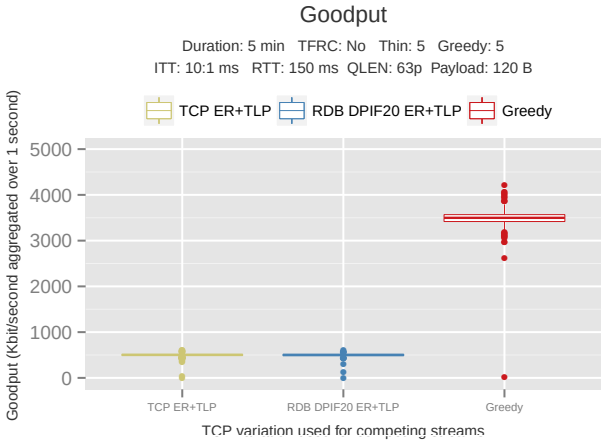
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
242	216	220	224	232	368	414	437	449	464	485	542	678	1110	1747
228	215	218	222	226	236	272	353	386	415	440	472	799	1043	1257
248	219	224	230	250	384	423	441	452	467	491	544	722	859	1151

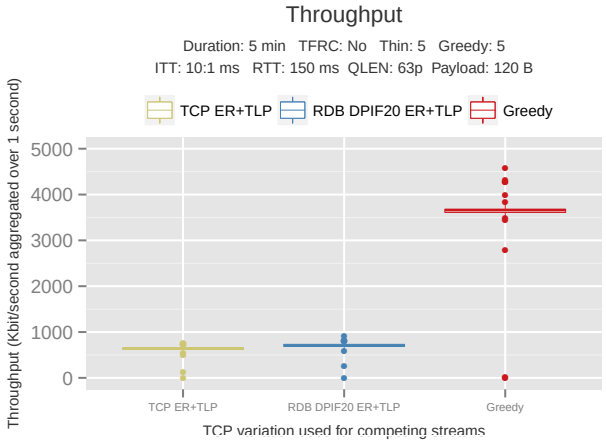
(c)

Figure A.3.3



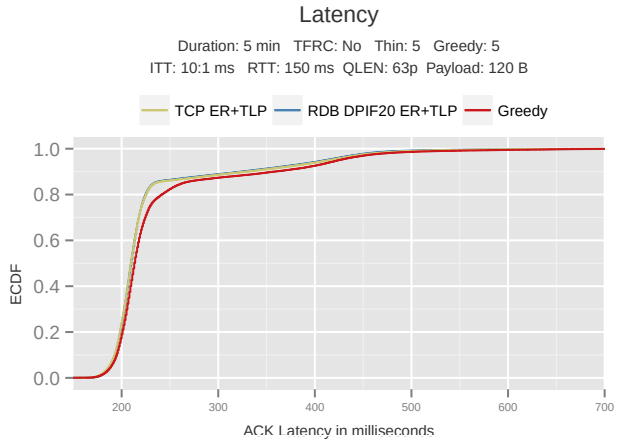
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	17 / 18 / 19	225/234/249	13/1/10/13/16/28
RDB DPIF20 ER+TLP	rdbsender	5	16 / 17 / 18	232/250/267	14/1/10/13/17/28
Greedy	zsender	5	15 / 16 / 17	1446/1447/1447	14/3/10/13/16/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
81.3 K	1089	1091	0	19.0 M	18.8 M	199.2 K	199.2 K	1.05 / 1.34	11307
84.4 K	1146	1061	7.7 K	21.1 M	18.5 M	2.5 M	208.4 K	1.25 / 1.36	11848
92.5 K	1581	1583	0	133.9 M	131.6 M	2.3 M	2.3 M	1.71 / 1.71	12995

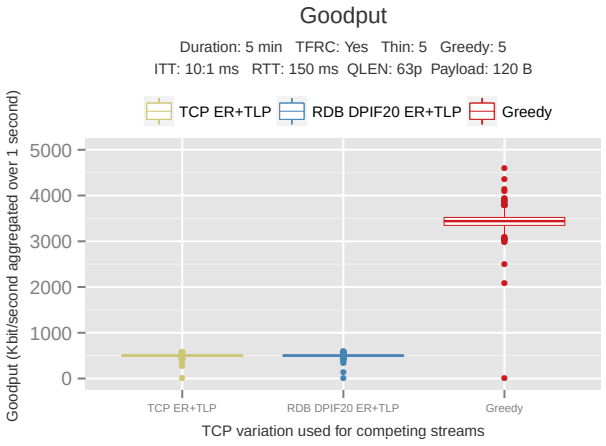
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
232	210	213	218	225	330	386	415	428	442	462	501	665	895	1150
232	210	213	218	225	324	381	411	424	438	455	487	718	986	1655
240	213	218	225	242	358	405	425	436	451	473	531	695	853	1152

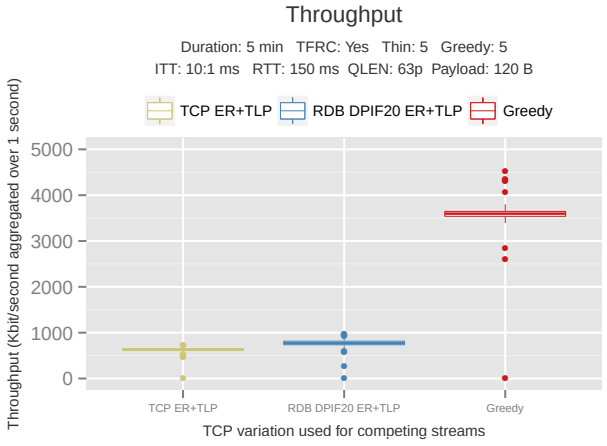
(c)

Figure A.3.4



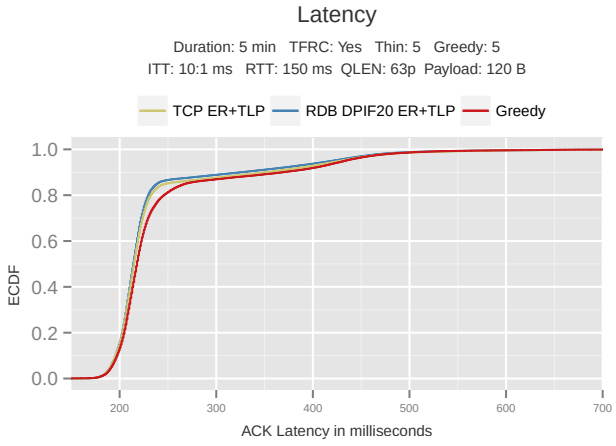
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	18 / 20 / 21	238/253/267	13/1/9/12/16/28
RDB DPIF20 ER+TLP	rdbsender	5	19 / 20 / 21	317/333/352	14/1/8/13/20/36
Greedy	zsender	5	16 / 16 / 17	1446/1447/1447	13/2/10/12/15/31

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
75.3 K	1129	1131	0	19.0 M	18.8 M	213.2 K	213.2 K	1.12 / 1.50	10976
73.0 K	1094	890	15.6 K	24.3 M	18.6 M	5.8 M	167.6 K	1.28 / 1.50	9406
91.4 K	1565	1567	0	132.2 M	129.9 M	2.3 M	2.3 M	1.71 / 1.71	12715

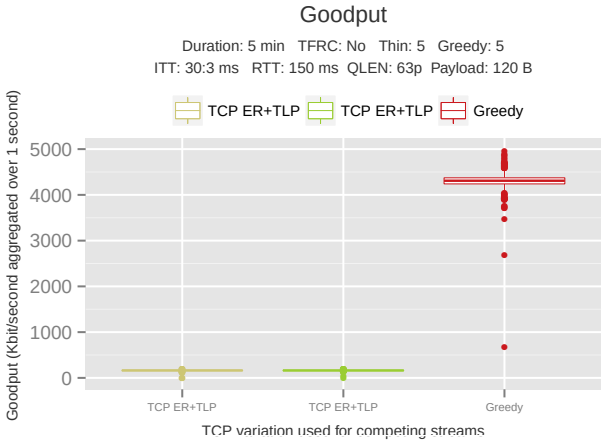
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
240	215	219	224	232	351	402	429	442	456	478	533	755	891	1153
236	214	218	222	230	325	387	420	435	450	470	515	723	1018	1239
245	218	223	230	246	369	413	433	444	458	478	519	701	966	1211

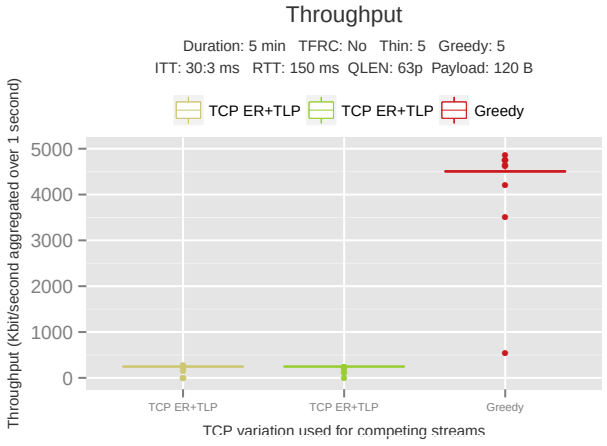
(c)

Figure A.3.5



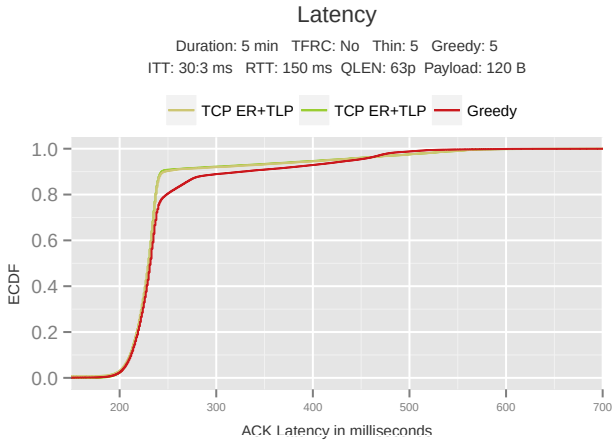
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 32	126/127/129	11/1/8/12/13/13
TCP ER+TLP	rdbsender	5	31 / 31 / 31	126/127/128	11/2/9/12/13/14
Greedy	zsender	5	11 / 13 / 14	1447/1447/1447	18/6/14/17/21/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.4 K	515	517	0	6.1 M	6.1 M	61.9 K	61.9 K	1.01 / 1.06	4150
47.7 K	505	507	0	6.1 M	6.0 M	61.2 K	61.2 K	1.01 / 1.06	3970
113.2 K	1117	1117	0	163.9 M	162.3 M	1.6 M	1.6 M	0.99 / 0.99	13799

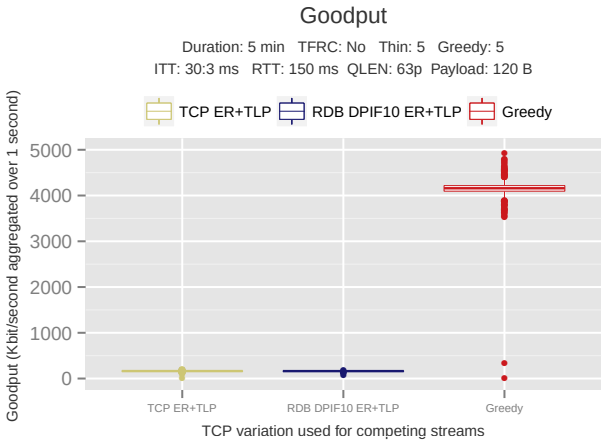
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
245	230	233	235	237	247	346	417	450	482	516	548	631	841	3232
245	230	233	235	237	243	336	412	448	482	516	548	633	834	1300
252	232	235	238	249	327	403	443	458	468	479	507	603	811	999

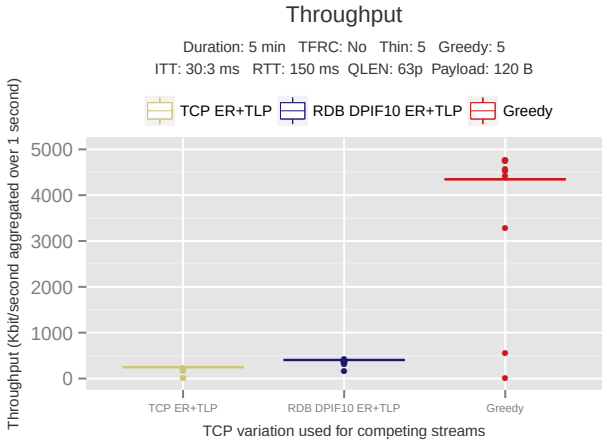
(c)

Figure A.3.6



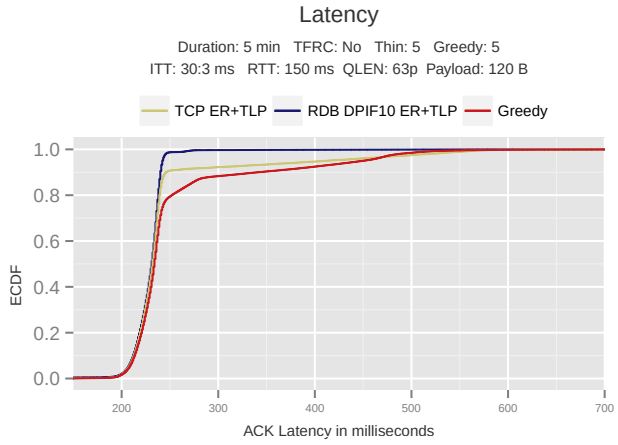
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 31	126/126/128	11/2/9/12/13/14
RDB DPIF10 ER+TLP	rdbsender	5	31 / 31 / 32	253/256/258	11/2/9/12/13/14
Greedy	zsender	5	12 / 14 / 14	1446/1447/1448	17/5/14/17/20/49

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.8 K	497	499	0	6.2 M	6.1 M	59.8 K	59.8 K	0.97 / 1.02	4029
47.1 K	438	22	47.0 K	12.0 M	6.0 M	6.0 M	3.0 K	0.90 / 0.93	124
109.4 K	1181	1183	0	158.4 M	156.6 M	1.7 M	1.7 M	1.08 / 1.08	13640

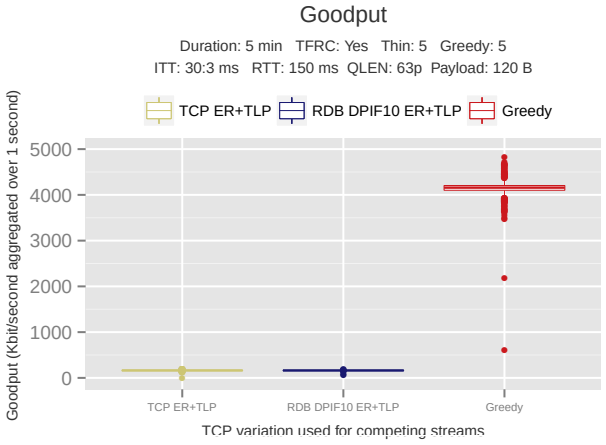
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
246	232	235	237	239	246	333	412	448	482	518	552	649	932	1153
229	231	234	236	238	240	241	242	243	244	246	264	511	626	692
255	234	237	240	252	341	411	450	464	474	487	514	676	798	1151

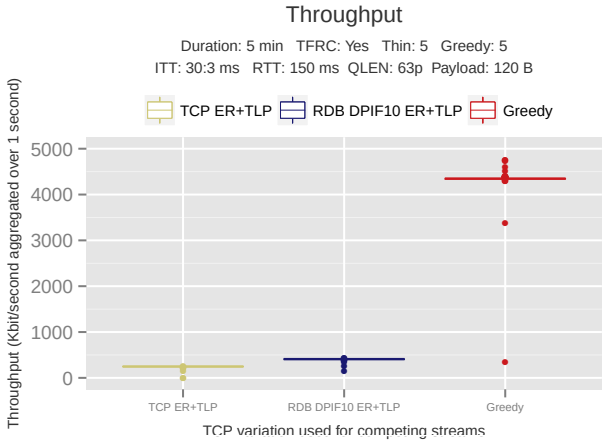
(c)

Figure A.3.7



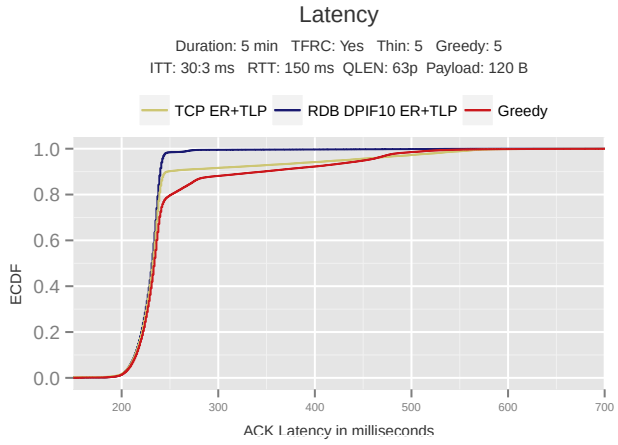
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 32	126/127/130	11/1/8/12/13/14
RDB DPIF10 ER+TLP	rdbsender	5	30 / 30 / 30	241/242/244	12/1/11/12/13/31
Greedy	zsender	5	13 / 13 / 15	1446/1447/1448	17/5/14/17/20/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.5 K	548	550	0	6.2 M	6.1 M	66.2 K	66.2 K	1.07 / 1.13	4245
49.8 K	511	28	49.7 K	12.0 M	6.0 M	6.0 M	3.4 K	1.02 / 1.03	234
109.3 K	1204	1204	0	158.2 M	156.5 M	1.7 M	1.7 M	1.10 / 1.10	13798

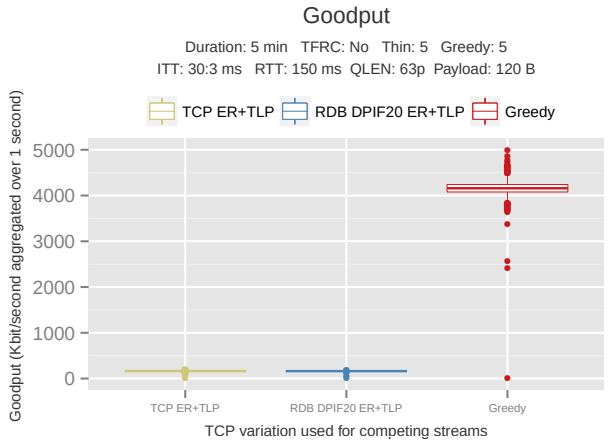
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
248	232	235	237	239	247	356	430	463	493	524	553	624	828	1254
231	231	234	236	238	240	241	242	243	244	246	268	530	625	680
256	234	236	239	251	346	417	453	464	473	486	515	650	914	999

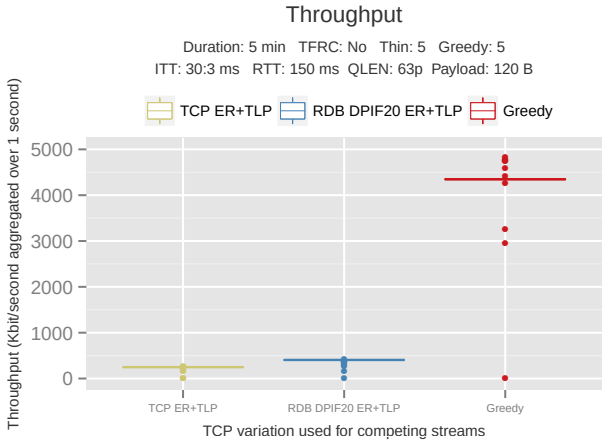
(c)

Figure A.3.8



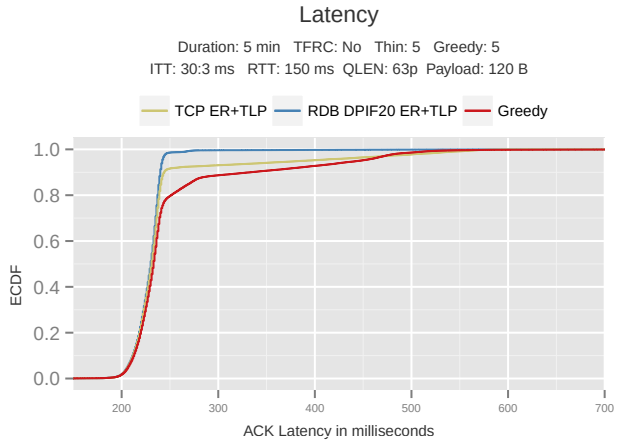
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	30 / 31 / 31	124/125/126	11/3/9/12/13/13
RDB DPIF20 ER+TLP	rdbsender	5	31 / 31 / 32	251/253/255	11/2/9/12/13/13
Greedy	zsender	5	13 / 13 / 15	1446/1447/1447	17/6/14/16/20/44

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
49.1 K	452	454	0	6.2 M	6.1 M	54.5 K	54.5 K	0.89 / 0.92	3611
47.3 K	437	25	47.0 K	12.0 M	6.0 M	6.0 M	2.8 K	0.90 / 0.92	163
109.6 K	1127	1129	0	158.6 M	156.9 M	1.6 M	1.6 M	1.03 / 1.03	13417

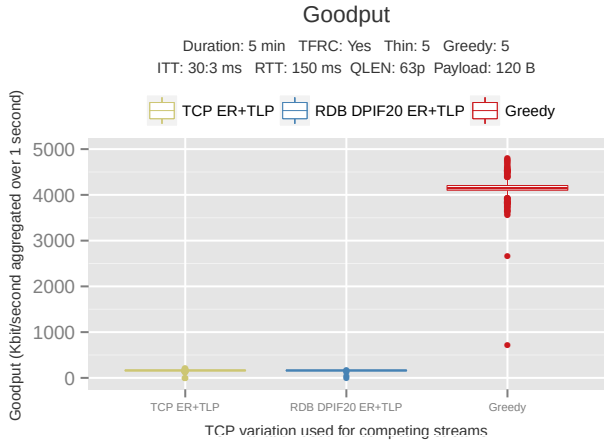
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
244	231	234	236	239	244	296	387	431	470	507	543	594	641	1152
229	230	233	235	237	240	241	242	243	244	246	266	516	791	1153
254	233	236	239	251	333	405	446	461	471	483	512	689	878	1152

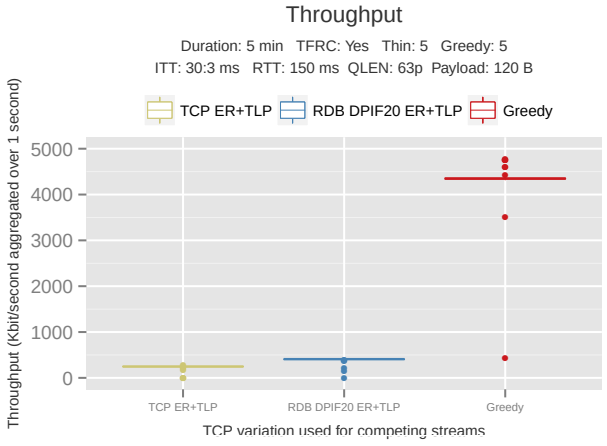
(c)

Figure A.3.9



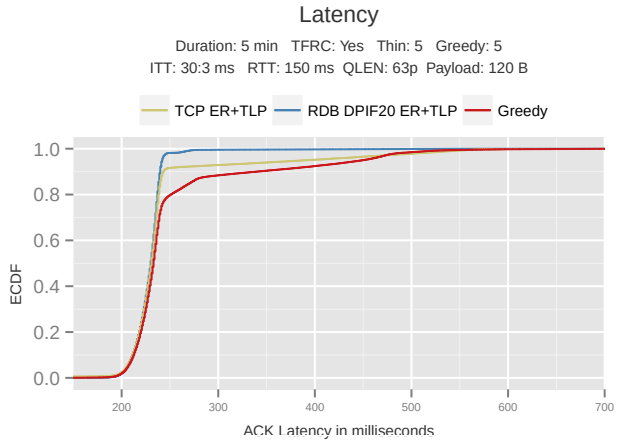
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 31	125/126/127	11/1/9/12/13/13
RDB DPIF20 ER+TLP	rdbsender	5	30 / 30 / 30	240/242/244	14/2/12/13/13/106
Greedy	zsender	5	13 / 13 / 14	1447/1447/1447	17/4/14/17/20/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.8 K	446	448	0	6.1 M	6.1 M	54.1 K	54.1 K	0.88 / 0.91	3746
49.3 K	487	28	49.0 K	12.0 M	6.0 M	6.0 M	3.1 K	0.98 / 0.99	237
109.3 K	1134	1134	0	158.3 M	156.6 M	1.6 M	1.6 M	1.04 / 1.04	13736

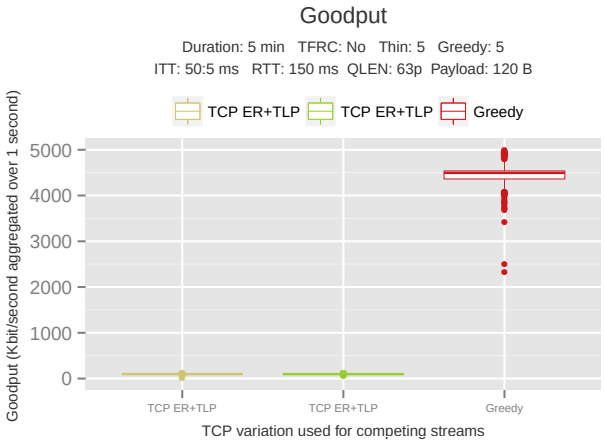
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
243	231	234	236	238	243	306	392	433	470	505	540	624	854	3148
229	230	233	235	237	240	241	242	243	244	247	268	512	594	1297
255	233	236	239	251	340	413	450	462	472	485	517	675	763	999

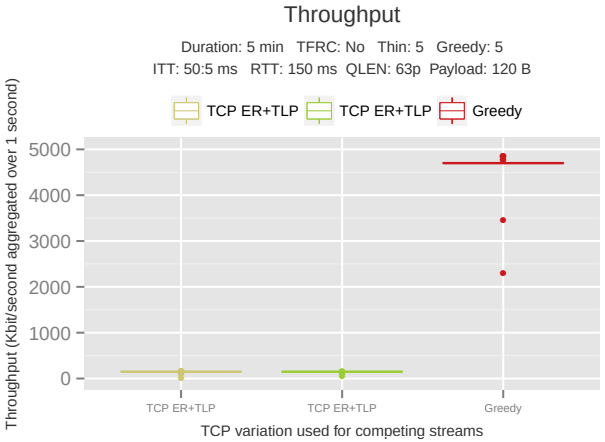
(c)

Figure A.3.10



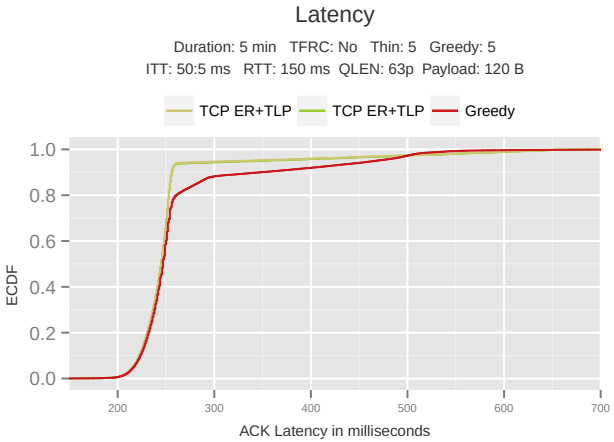
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	121/121/122	9/1/9/10/10/10
TCP ER+TLP	rdbsender	5	50 / 50 / 50	121/121/122	9/1/9/10/10/10
Greedy	zsender	5	11 / 12 / 14	1446/1446/1447	20/7/16/19/23/50

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.2 K	250	252	0	3.7 M	3.6 M	30.0 K	30.0 K	0.82 / 0.83	1616
29.8 K	264	264	0	3.6 M	3.6 M	31.7 K	31.7 K	0.87 / 0.88	1669
118.2 K	1056	1056	0	170.9 M	169.4 M	1.5 M	1.5 M	0.89 / 0.89	14357

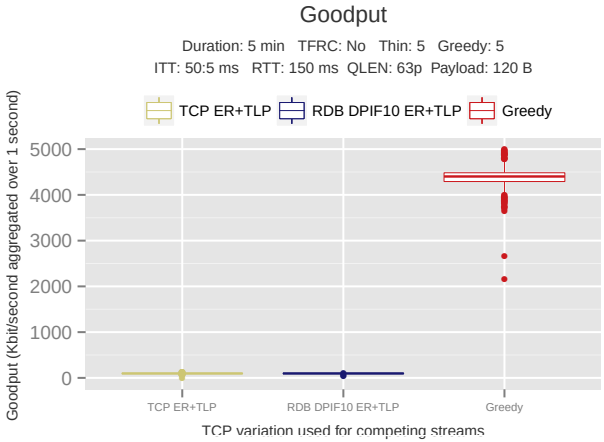
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
255	245	248	251	253	256	258	332	402	476	541	601	668	780	1273
255	245	248	251	253	256	259	348	419	486	546	604	663	736	787
268	247	250	254	261	348	425	467	485	497	508	538	727	806	999

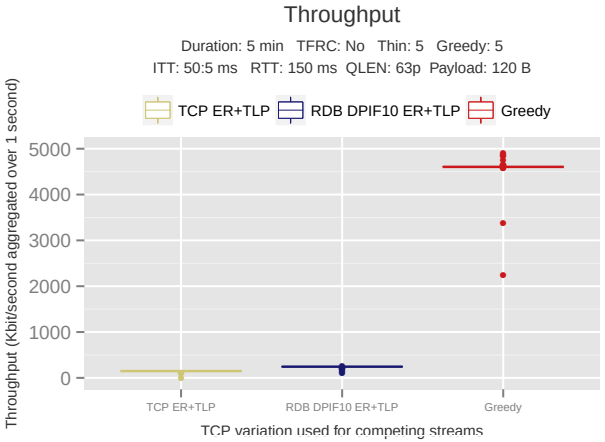
(c)

Figure A.3.11



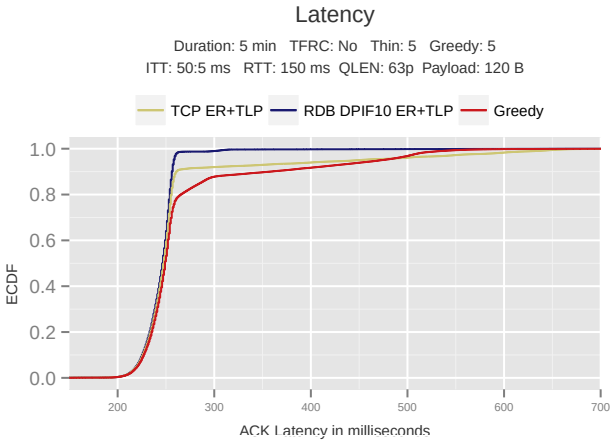
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	122/122/123	9/1/7/9/10/11
RDB DPIF10 ER+TLP	rdbsender	5	50 / 50 / 51	243/244/247	9/1/8/10/10/11
Greedy	zsender	5	12 / 13 / 13	1446/1447/1447	20/7/16/19/23/45

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.1 K	362	362	0	3.7 M	3.6 M	43.6 K	43.6 K	1.18 / 1.20	2384
29.5 K	300	14	29.4 K	7.2 M	3.6 M	3.6 M	2.2 K	1.02 / 1.02	86
115.9 K	1101	1101	0	167.6 M	166.1 M	1.6 M	1.6 M	0.95 / 0.95	14409

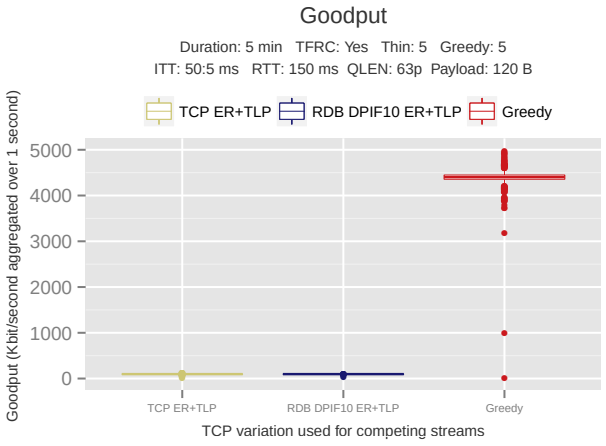
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
264	248	251	253	256	261	351	449	497	543	590	632	695	964	1015
245	247	249	252	254	256	257	258	259	260	261	300	567	692	1226
271	249	252	256	265	358	430	473	490	502	512	535	656	838	999

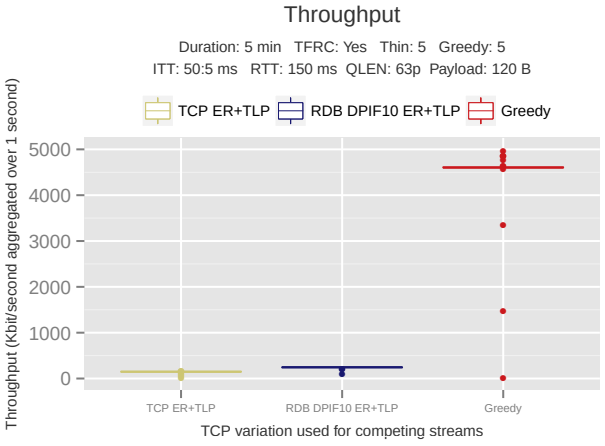
(c)

Figure A.3.12



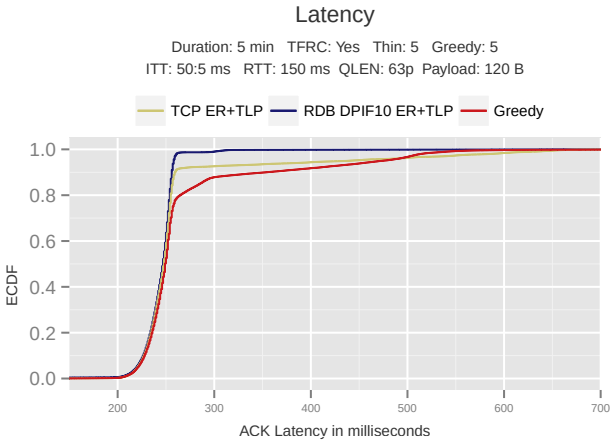
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	121/122/123	9/1/8/10/10/10
RDB DPIF10 ER+TLP	rdbsender	5	49 / 50 / 50	240/240/241	11/1/10/10/11/24
Greedy	zsender	5	12 / 13 / 14	1446/1447/1447	20/8/16/19/23/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.1 K	346	348	0	3.7 M	3.6 M	41.6 K	41.6 K	1.13 / 1.15	2153
29.9 K	319	13	29.9 K	7.2 M	3.6 M	3.6 M	1.6 K	1.06 / 1.07	72
115.8 K	1115	1117	0	167.6 M	166.0 M	1.6 M	1.6 M	0.96 / 0.96	14183

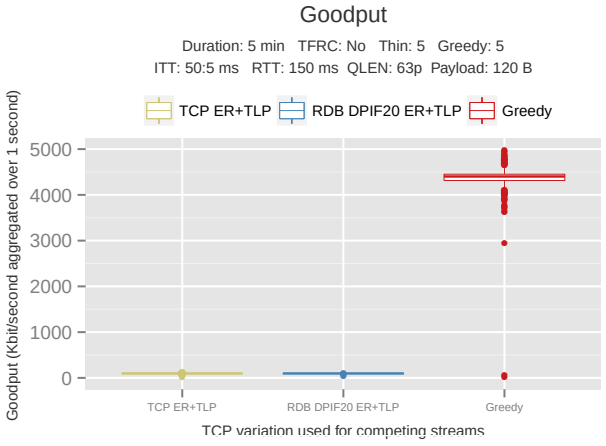
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
263	248	251	253	255	259	321	438	489	539	585	631	691	969	1152
245	247	250	252	254	256	257	258	259	260	261	300	552	650	662
271	249	252	255	265	354	430	474	492	503	514	540	715	951	1152

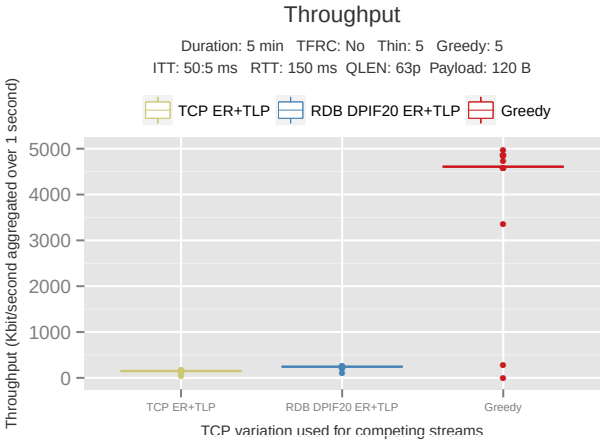
(c)

Figure A.3.13



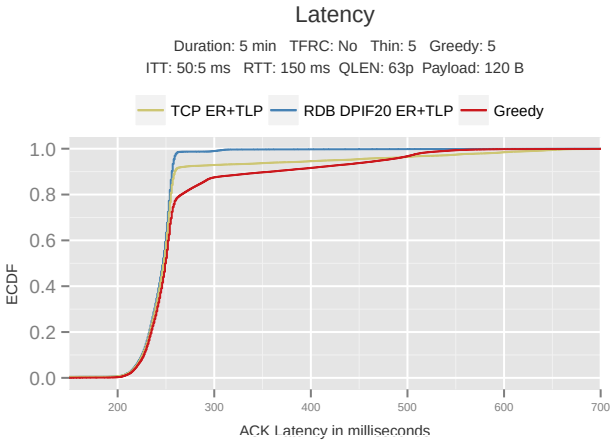
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	122/122/123	9/1/8/10/10/10
RDB DPIF20 ER+TLP	rdbsender	5	50 / 50 / 51	242/243/245	9/1/8/10/10/10
Greedy	zsender	5	11 / 13 / 14	1446/1447/1447	20/4/15/19/24/45

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.2 K	344	345	0	3.7 M	3.6 M	41.4 K	41.4 K	1.12 / 1.14	2097
29.6 K	307	16	29.5 K	7.2 M	3.6 M	3.6 M	1.9 K	1.03 / 1.04	83
115.8 K	1133	1135	0	167.5 M	165.8 M	1.6 M	1.6 M	0.98 / 0.98	14439

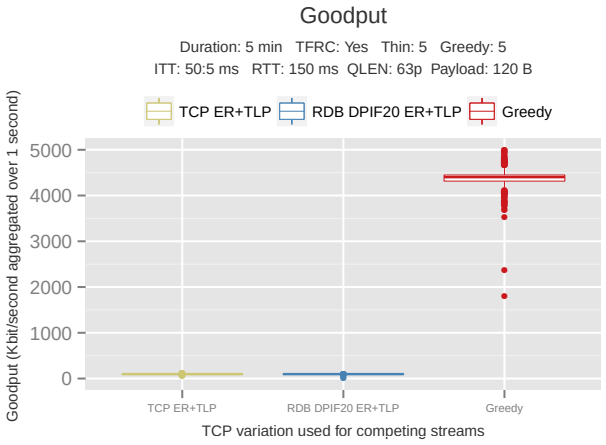
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
262	248	251	253	255	259	307	427	481	531	581	630	696	985	1101
245	247	250	252	254	256	257	258	259	260	261	300	580	699	760
271	250	253	256	266	359	433	475	491	503	513	538	698	794	1151

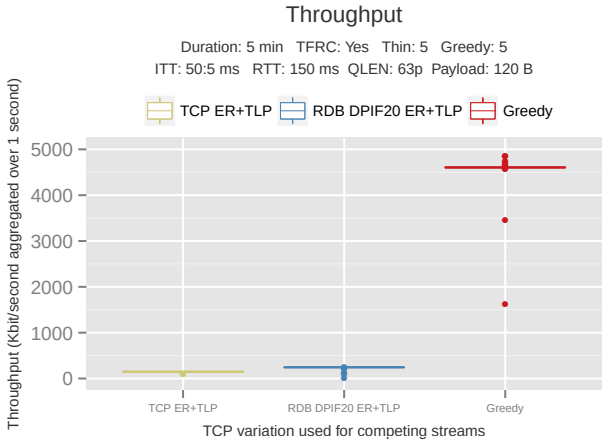
(c)

Figure A.3.14



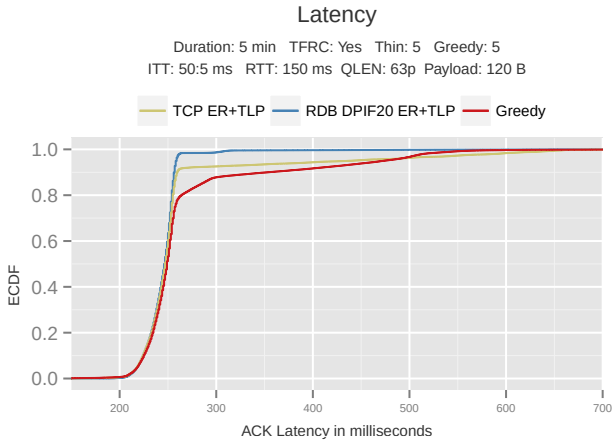
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	50 / 50 / 50	122/122/123	9/1/8/10/10/11
RDB DPIF20 ER+TLP	rdbsender	5	50 / 50 / 50	240/241/242	12/1/10/10/10/88
Greedy	zsender	5	11 / 13 / 14	1447/1447/1447	20/4/16/19/23/46

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.1 K	354	355	0	3.7 M	3.6 M	42.7 K	42.7 K	1.16 / 1.17	2163
29.7 K	358	22	29.7 K	7.2 M	3.6 M	3.6 M	2.4 K	1.20 / 1.20	124
115.8 K	1089	1089	0	167.5 M	166.0 M	1.6 M	1.6 M	0.94 / 0.94	14278

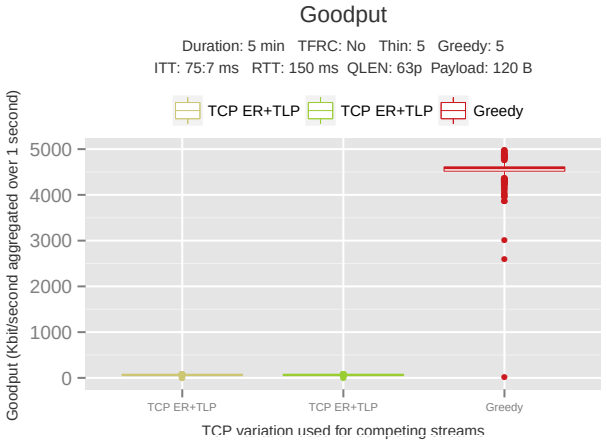
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
262	247	250	253	255	260	327	437	489	540	588	633	710	1001	1348
245	246	250	252	254	257	258	259	259	260	262	306	598	672	1272
270	249	252	256	264	353	432	473	491	503	514	544	708	809	999

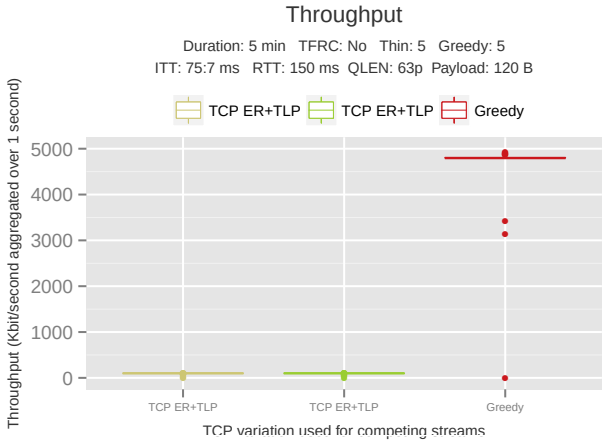
(c)

Figure A.3.15



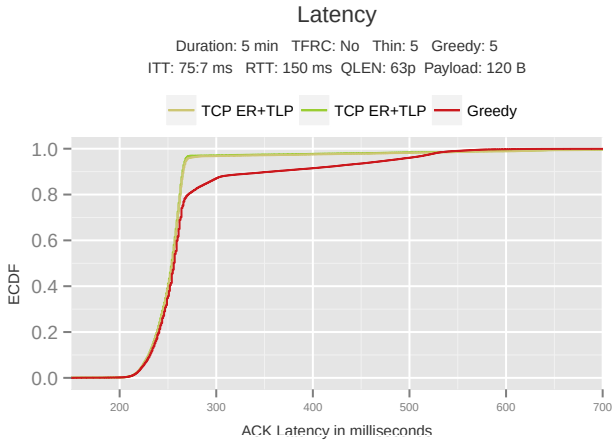
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 75	120/120/120	8/1/8/8/8/10
TCP ER+TLP	rdbsender	5	74 / 74 / 75	120/120/120	8/1/8/8/8/10
Greedy	zsender	5	11 / 12 / 13	1447/1447/1448	21/6/17/20/24/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.3 K	123	125	0	2.4 M	2.4 M	14.8 K	14.8 K	0.61 / 0.61	600
19.9 K	104	106	0	2.4 M	2.4 M	12.5 K	12.5 K	0.52 / 0.52	543
120.6 K	1058	1057	0	174.6 M	173.0 M	1.5 M	1.5 M	0.88 / 0.88	14488

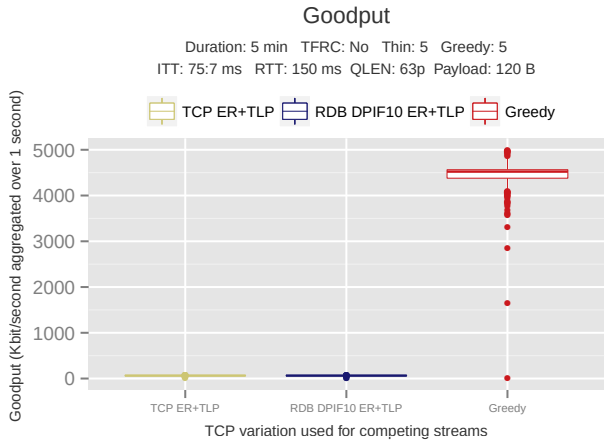
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
260	254	257	260	263	266	267	269	272	328	477	615	717	1011	1152
258	254	257	260	262	265	267	268	269	283	436	600	713	1002	1151
277	256	260	263	271	357	436	480	499	515	526	548	637	803	1151

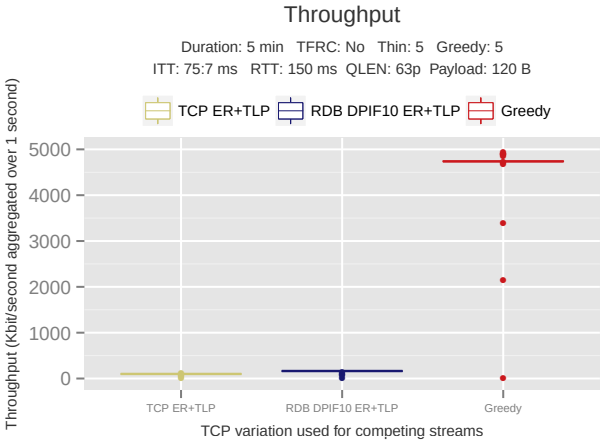
(c)

Figure A.3.16



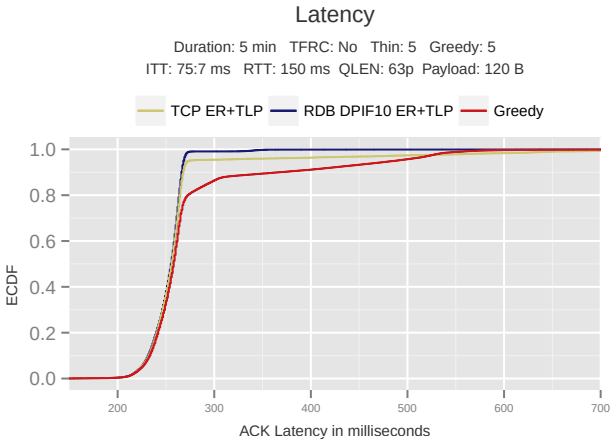
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 75	120/120/121	8/1/8/8/8/10
RDB DPIF10 ER+TLP	rdbsender	5	75 / 75 / 75	240/240/241	8/1/8/8/8/10
Greedy	zsender	5	12 / 12 / 13	1446/1446/1448	21/1/17/20/24/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.2 K	183	187	0	2.4 M	2.4 M	22.2 K	22.2 K	0.90 / 0.90	832
19.8 K	158	10	19.8 K	4.8 M	2.4 M	2.4 M	960	0.80 / 0.80	20
119.0 K	1038	1040	0	172.2 M	170.7 M	1.5 M	1.5 M	0.87 / 0.87	14737

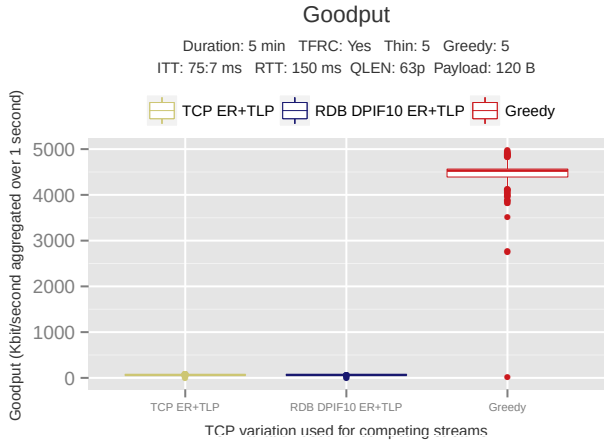
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
265	256	259	262	265	268	270	274	355	469	562	649	746	1018	1150
253	255	259	261	264	266	267	268	269	270	271	275	440	755	1152
280	258	262	265	273	367	444	487	505	520	531	555	750	2546	2726

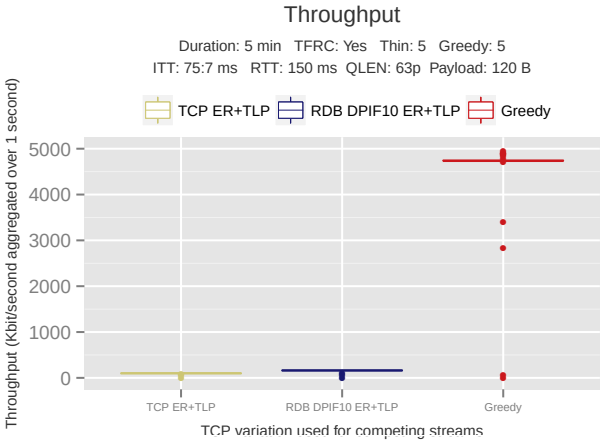
(c)

Figure A.3.17



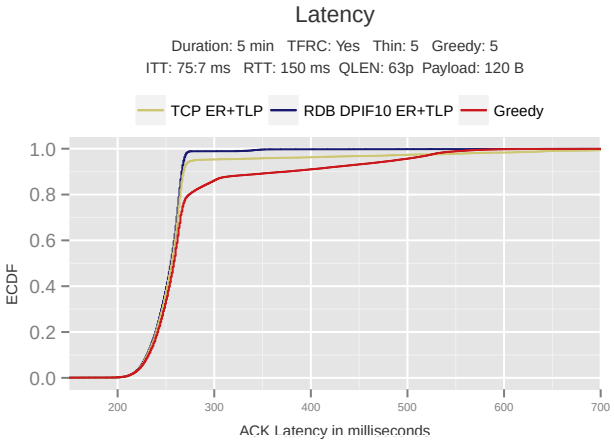
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 74	120/120/120	8/1/8/8/8/10
RDB DPIF10 ER+TLP	rdbsender	5	75 / 75 / 75	240/240/240	12/1/9/12/14/34
Greedy	zsender	5	12 / 12 / 13	1446/1447/1447	21/7/17/20/25/49

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.3 K	180	182	0	2.4 M	2.4 M	21.6 K	21.6 K	0.88 / 0.89	880
19.9 K	179	14	19.8 K	4.8 M	2.4 M	2.4 M	1.4 K	0.90 / 0.90	42
119.1 K	1050	1052	0	172.3 M	170.8 M	1.5 M	1.5 M	0.88 / 0.88	15020

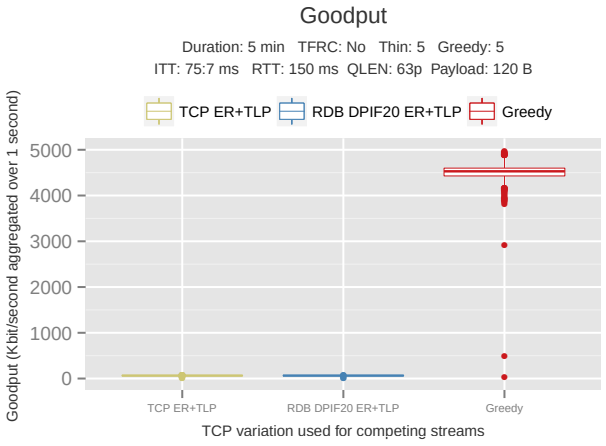
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
265	256	259	262	265	268	271	281	367	475	563	648	728	1012	1151
253	255	259	261	264	267	268	269	269	270	271	328	636	1013	1151
279	258	262	265	274	374	446	489	505	520	531	554	664	787	1151

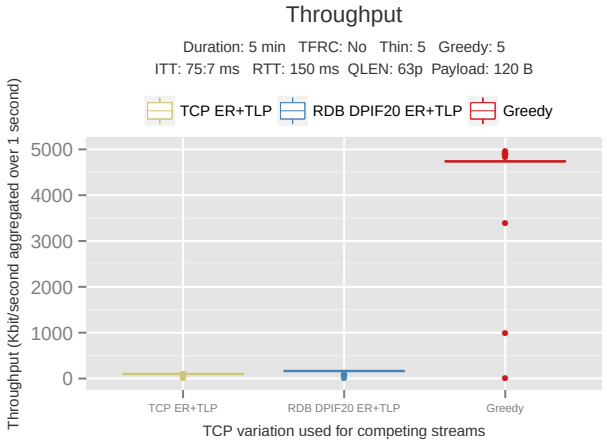
(c)

Figure A.3.18



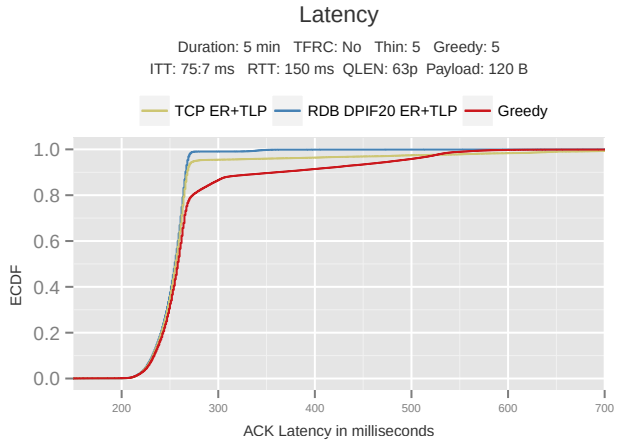
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 75	120/120/120	8/1/8/8/8/10
RDB DPIF20 ER+TLP	rdbsender	5	75 / 75 / 75	240/241/242	7/1/8/8/8/10
Greedy	zsender	5	12 / 12 / 14	1446/1447/1447	21/7/17/20/24/47

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.3 K	174	177	0	2.4 M	2.4 M	21.0 K	21.0 K	0.86 / 0.86	834
19.8 K	163	7	19.8 K	4.8 M	2.4 M	2.4 M	600	0.82 / 0.82	23
118.9 K	1000	1002	0	172.0 M	170.6 M	1.4 M	1.4 M	0.84 / 0.84	14328

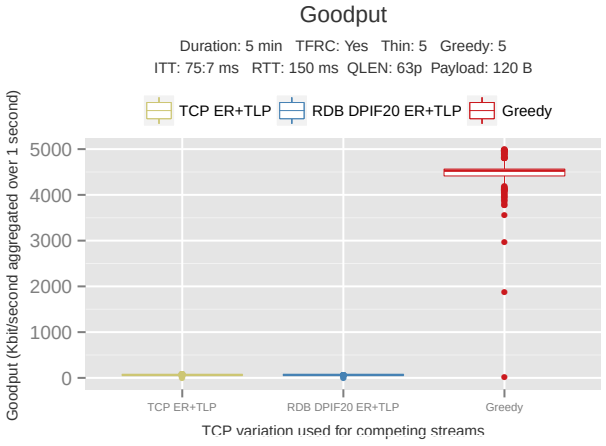
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
265	256	259	262	265	268	270	277	355	466	565	650	768	995	1152
253	255	259	261	264	266	267	268	269	270	271	275	477	766	1151
279	258	262	265	274	360	439	484	503	518	530	553	673	854	1151

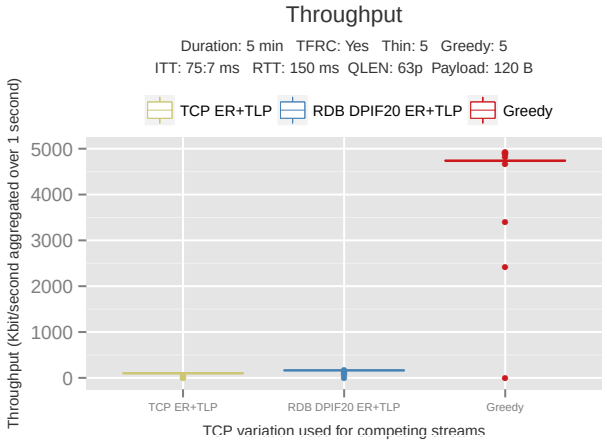
(c)

Figure A.3.19



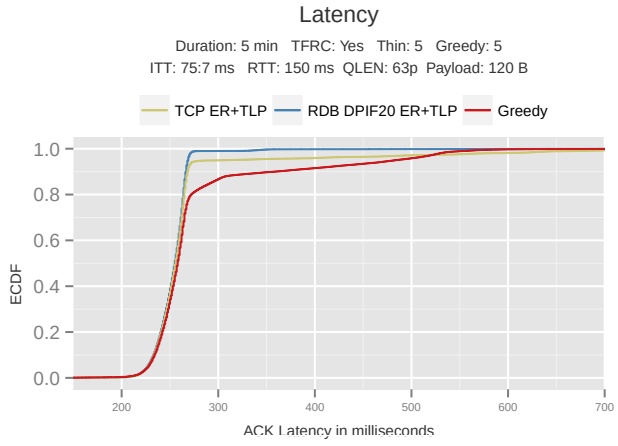
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	74 / 74 / 74	120/121/121	7/1/8/8/8/10
RDB DPIF20 ER+TLP	rdbsender	5	75 / 75 / 75	240/240/240	16/1/11/13/17/85
Greedy	zsender	5	11 / 12 / 13	1446/1447/1447	21/1/17/20/24/49

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
20.3 K	202	204	0	2.4 M	2.4 M	24.2 K	24.2 K	0.99 / 1.00	938
19.9 K	162	12	19.8 K	4.8 M	2.4 M	2.4 M	1.2 K	0.82 / 0.81	38
119.0 K	1023	1025	0	172.2 M	170.7 M	1.5 M	1.5 M	0.86 / 0.86	14277

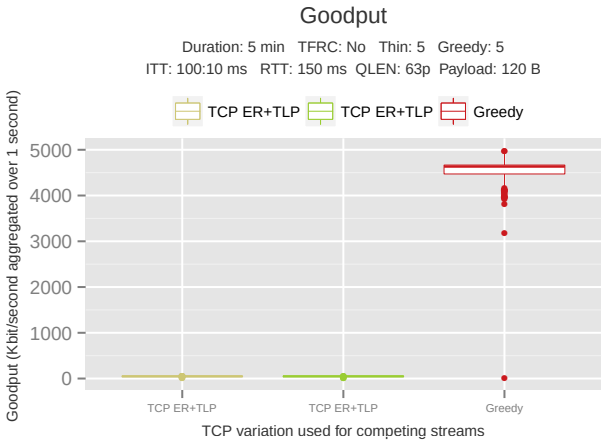
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
267	256	259	262	265	268	271	308	403	491	576	672	783	1170	1313
254	255	259	261	264	267	268	269	269	270	271	301	610	797	1151
279	258	262	265	272	359	439	484	504	518	530	554	711	1715	1935

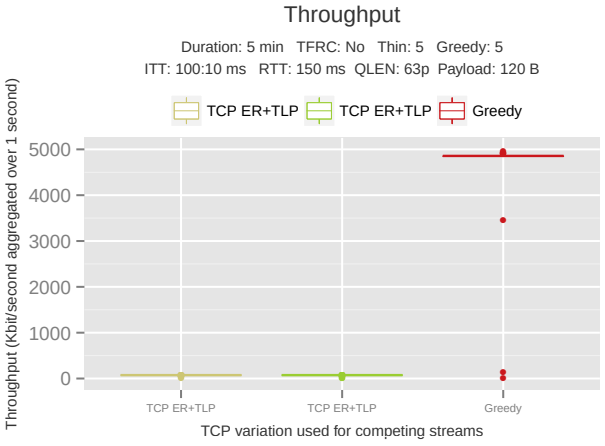
(c)

Figure A.3.20



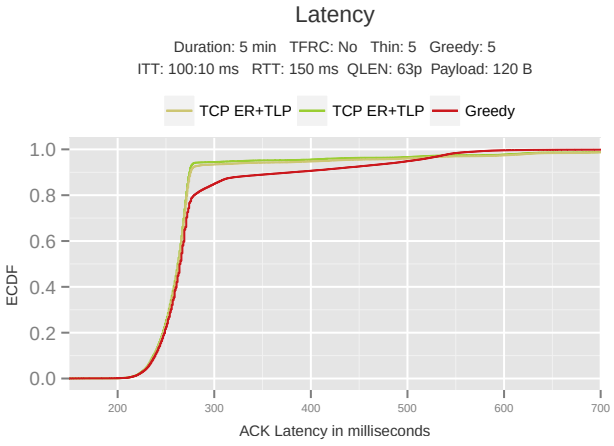
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	98 / 99 / 99	120/120/120	6/1/6/7/7/10
TCP ER+TLP	rdbsender	5	99 / 99 / 99	120/120/120	7/1/7/7/7/10
Greedy	zsender	5	11 / 12 / 13	1446/1447/1447	22/7/17/21/25/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
15.2 K	225	231	0	1.8 M	1.8 M	27.5 K	27.5 K	1.47 / 1.47	843
15.0 K	185	194	0	1.8 M	1.8 M	22.8 K	22.8 K	1.22 / 1.23	688
121.6 K	1074	1076	0	176.0 M	174.5 M	1.6 M	1.6 M	0.88 / 0.88	15202

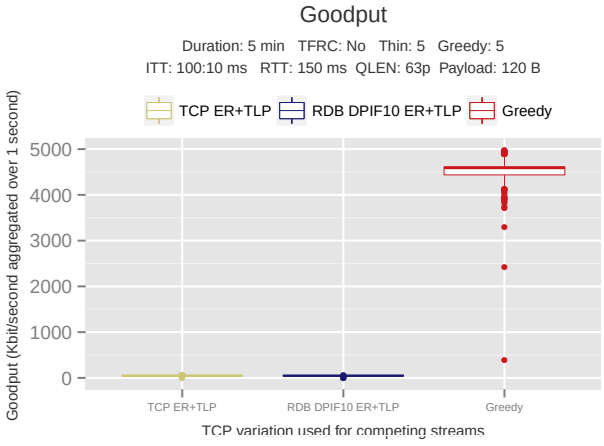
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
277	263	266	269	272	276	286	417	499	557	624	711	829	1095	1151
274	262	266	269	271	274	276	333	425	517	612	707	822	1043	1151
287	265	269	271	279	381	461	503	519	532	544	568	749	835	1151

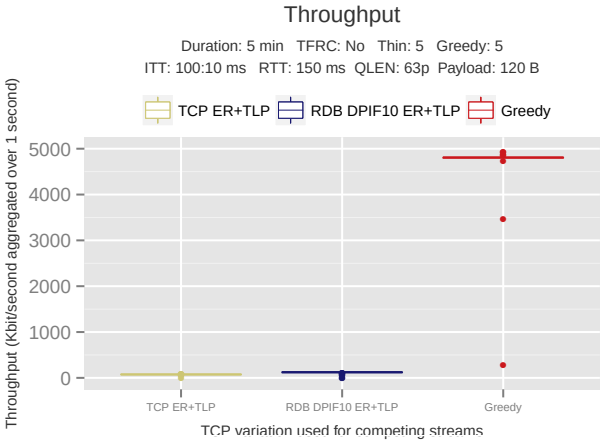
(c)

Figure A.3.21



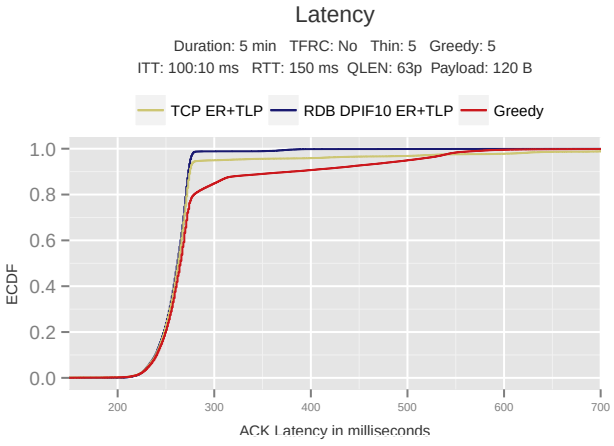
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	98 / 98 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF10 ER+TLP	rdbsender	5	100 / 100 / 100	240/240/240	7/1/7/7/7/10
Greedy	zsender	5	12 / 12 / 13	1446/1447/1447	22/7/17/21/26/48

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
15.3 K	179	189	0	1.8 M	1.8 M	22.7 K	22.7 K	1.17 / 1.17	628
14.8 K	150	11	14.8 K	3.6 M	1.8 M	1.8 M	1.1 K	1.00 / 1.01	22
120.5 K	1071	1071	0	174.3 M	172.8 M	1.6 M	1.6 M	0.89 / 0.89	14835

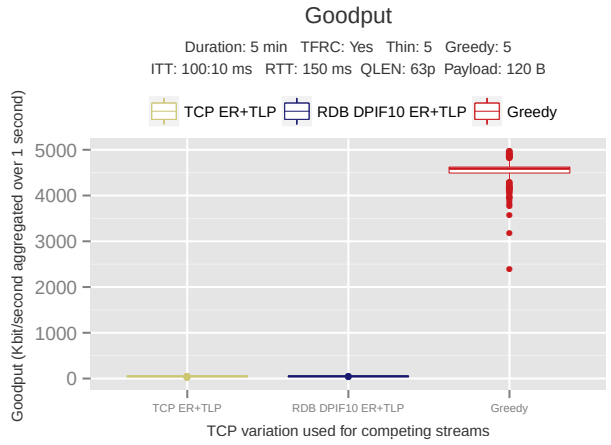
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
274	263	266	269	272	275	277	305	410	511	613	710	845	909	949
261	262	265	268	271	273	274	275	276	277	278	356	581	1060	1278
287	265	269	272	279	380	458	501	520	534	545	571	735	867	999

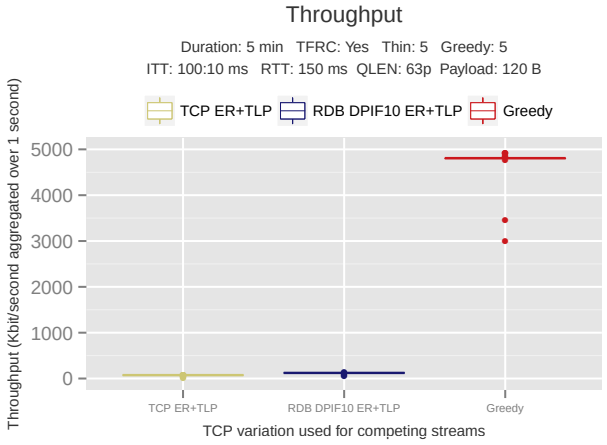
(c)

Figure A.3.22



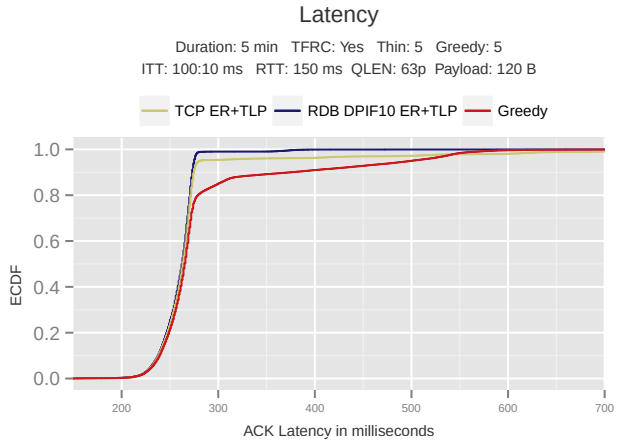
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	99 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF10 ER+TLP	rdbsender	5	99 / 99 / 100	240/240/240	12/2/9/10/13/28
Greedy	zsender	5	11 / 12 / 13	1446/1447/1447	22/7/17/21/25/53

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
15.2 K	154	158	0	1.8 M	1.8 M	18.7 K	18.7 K	1.01 / 1.01	570
14.9 K	137	5	14.9 K	3.6 M	1.8 M	1.8 M	600	0.92 / 0.92	14
120.7 K	1039	1039	0	174.6 M	173.1 M	1.5 M	1.5 M	0.86 / 0.86	14872

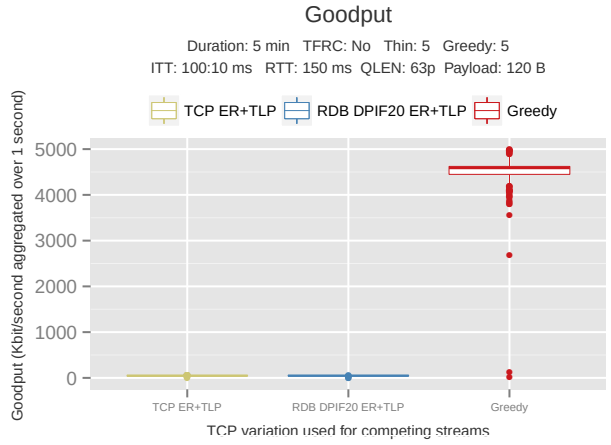
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
272	263	267	269	272	275	277	281	344	468	593	697	824	1222	1283
260	263	266	268	271	273	274	275	276	276	277	291	398	771	840
286	265	269	272	279	375	454	500	519	534	545	568	715	822	999

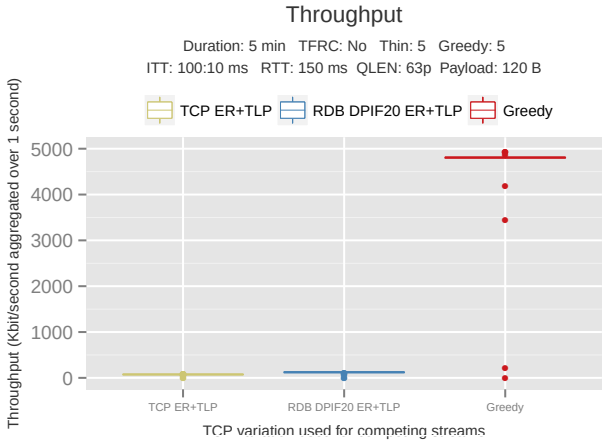
(c)

Figure A.3.23



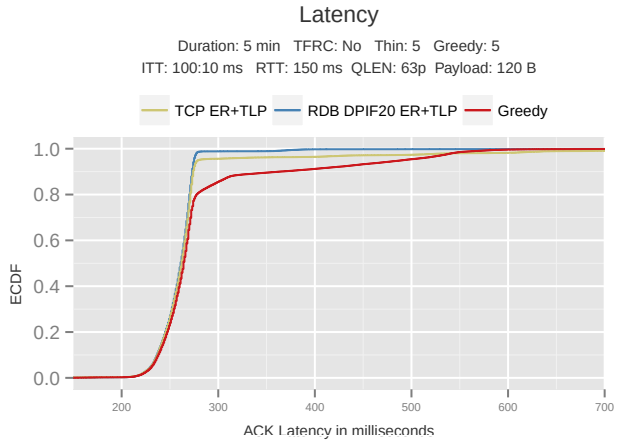
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	99 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF20 ER+TLP	rdbsender	5	100 / 100 / 100	240/240/240	7/1/7/7/7/10
Greedy	zsender	5	12 / 12 / 13	1446/1447/1447	21/9/17/21/25/45

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
15.2 K	145	152	0	1.8 M	1.8 M	18.0 K	18.0 K	0.95 / 0.95	554
14.9 K	140	16	14.9 K	3.6 M	1.8 M	1.8 M	1.4 K	0.93 / 0.94	33
120.3 K	975	977	0	174.1 M	172.7 M	1.4 M	1.4 M	0.81 / 0.81	14355

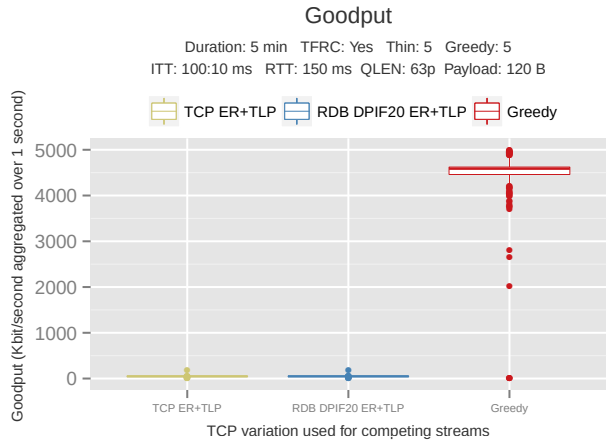
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
270	262	266	269	271	274	276	279	326	433	542	682	787	1025	1152
260	261	265	268	270	273	274	275	276	276	277	359	707	1033	1152
285	264	268	272	278	363	445	492	513	529	542	567	725	850	1151

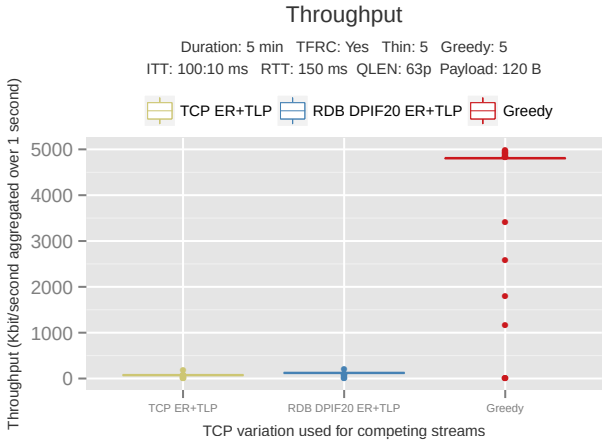
(c)

Figure A.3.24



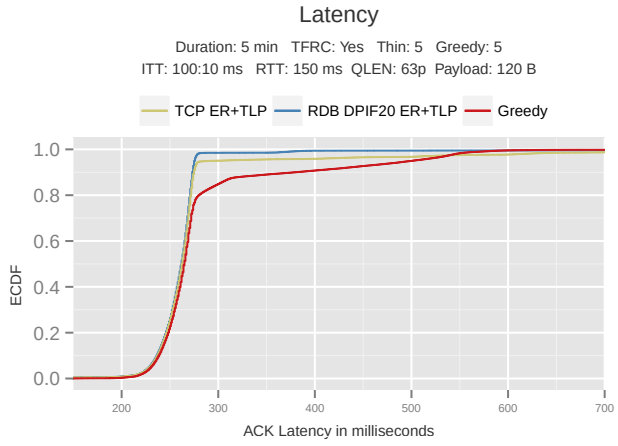
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	99 / 100 / 100	121/121/122	7/1/7/7/7/10
RDB DPIF20 ER+TLP	rdbsender	5	100 / 100 / 100	240/240/240	11/1/8/10/12/30
Greedy	zsender	5	12 / 12 / 13	1447/1447/1447	22/1/17/21/25/4

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
15.1 K	162	180	0	1.8 M	1.8 M	21.6 K	21.6 K	1.06 / 1.07	587
14.8 K	148	32	14.7 K	3.6 M	1.8 M	1.8 M	3.8 K	0.99 / 1.00	56
119.4 K	1082	1095	0	172.7 M	171.2 M	1.6 M	1.6 M	0.91 / 0.91	14813

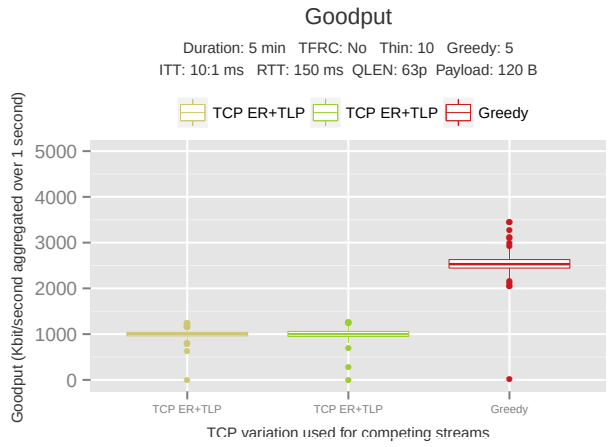
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
280	263	266	269	272	275	277	295	409	514	614	718	3428	3691	3724
269	262	265	268	271	273	275	275	276	277	279	373	3443	3711	3748
290	265	269	272	280	380	457	501	519	534	546	571	3532	3875	3988

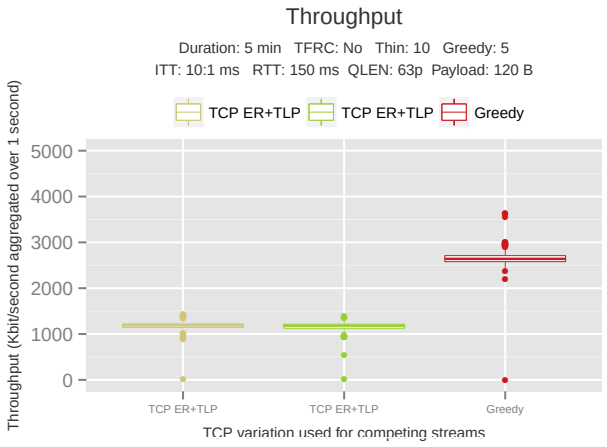
(c)

Figure A.3.25



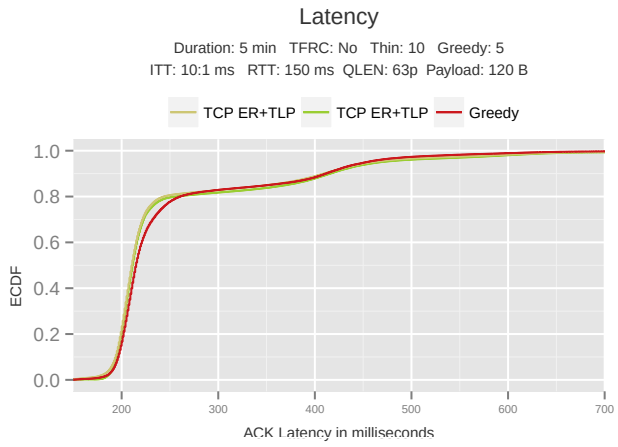
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	27 / 28 / 30	348/368/395	9/1/6/9/12/25
TCP ER+TLP	rdbsender	10	28 / 30 / 32	370/395/422	9/1/6/8/11/27
Greedy	zsender	5	21 / 22 / 22	1447/1447/1447	10/1/8/10/12/34

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
105.3 K	2962	2969	0	38.8 M	37.7 M	1.1 M	1.1 M	2.72 / 2.81	18963
97.2 K	3004	3011	0	38.3 M	37.1 M	1.2 M	1.2 M	3.07 / 3.09	18234
68.0 K	1903	1905	0	98.5 M	95.7 M	2.8 M	2.8 M	2.80 / 2.80	11641

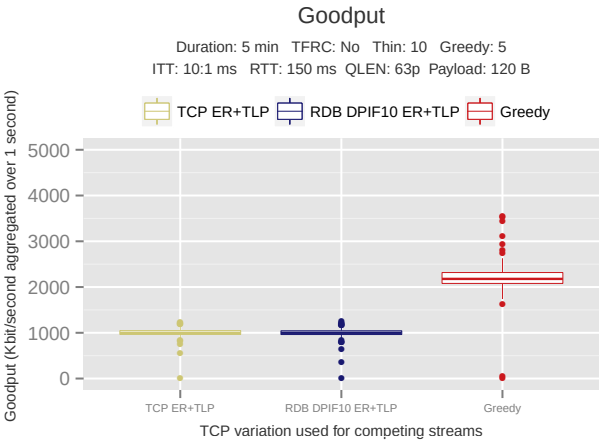
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
251	211	215	222	244	410	434	458	478	518	582	638	1035	1616	2237
256	211	216	223	257	416	440	468	496	551	599	652	1230	3013	3275
254	215	221	232	261	411	433	453	467	489	534	607	877	1831	2415

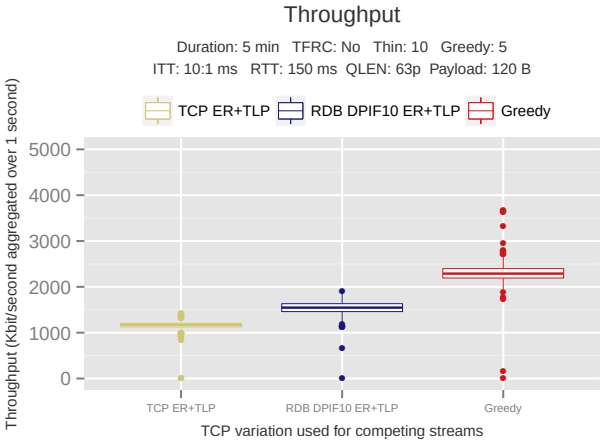
(c)

Figure A.3.26



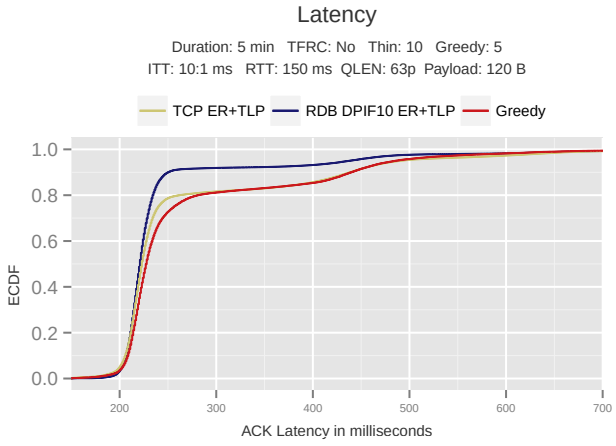
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	28 / 32 / 35	369/410/458	9/1/6/8/11/27
RDB DPIF10 ER+TLP	rdbsender	10	34 / 36 / 39	626/663/698	8/1/5/7/10/25
Greedy	zsender	5	22 / 25 / 28	1447/1447/1447	10/1/7/9/11/36

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
94.8 K	2904	2911	0	38.8 M	37.7 M	1.1 M	1.1 M	2.73 / 3.06	17262
81.1 K	2877	1276	56.1 K	53.7 M	37.1 M	16.6 M	959.9 K	3.54 / 3.55	6002
59.4 K	1824	1826	0	86.0 M	83.4 M	2.6 M	2.6 M	3.07 / 3.07	10418

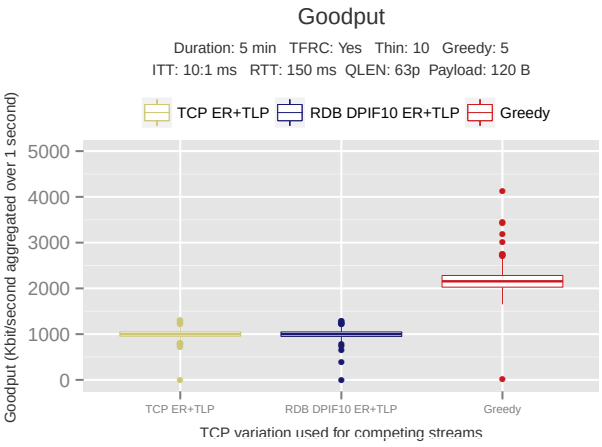
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
269	223	228	234	262	439	463	490	524	585	629	678	1352	3472	3558
245	221	224	229	234	250	394	438	454	474	554	664	1276	2006	2798
271	228	234	245	281	440	462	485	505	535	585	651	944	1776	2176

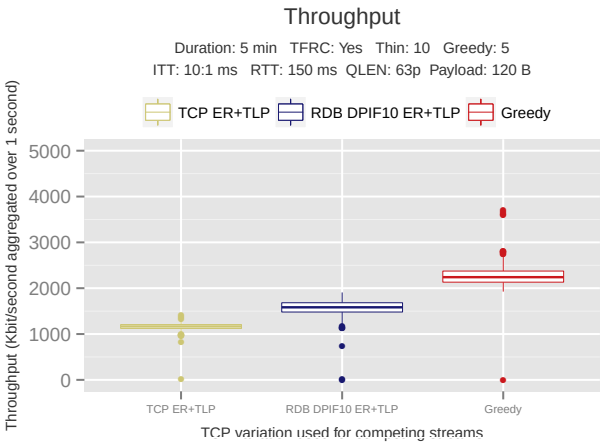
(c)

Figure A.3.27



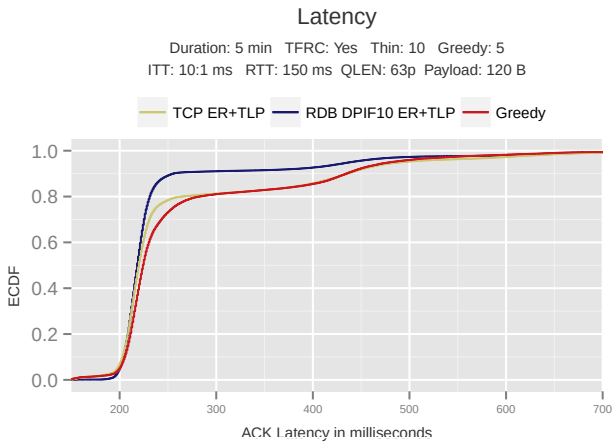
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	28 / 32 / 37	366/416/484	9/1/6/8/11/24
RDB DPIF10 ER+TLP	rdbsender	10	30 / 33 / 36	569/603/650	9/1/6/8/11/59
Greedy	zsender	5	25 / 25 / 27	1446/1447/1447	10/1/7/9/11/34

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
94.1 K	3050	3057	0	39.0 M	37.7 M	1.3 M	1.3 M	3.23 / 3.24	17211
90.6 K	3454	1499	65.7 K	54.6 M	37.1 M	17.5 M	1.0 M	3.77 / 3.81	7479
58.8 K	1830	1832	0	85.1 M	82.5 M	2.6 M	2.6 M	3.11 / 3.11	10459

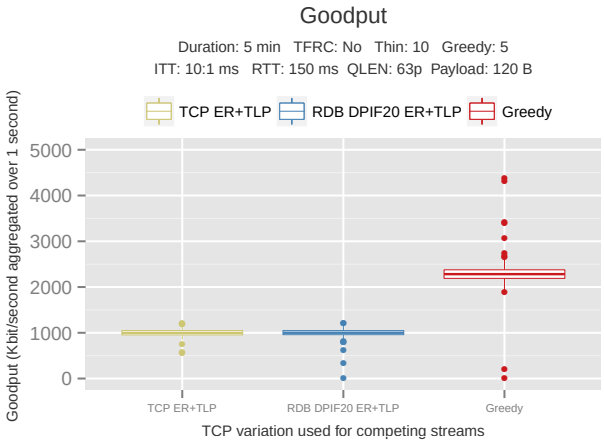
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
266	220	224	231	264	436	460	492	529	588	630	686	1191	1884	2809
246	218	222	225	231	255	408	439	456	484	601	672	1443	3579	6123
269	224	230	243	283	435	456	481	500	532	585	649	1057	3926	4142

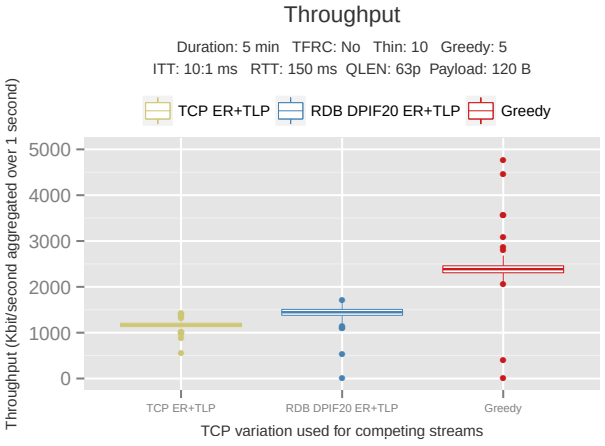
(c)

Figure A.3.28



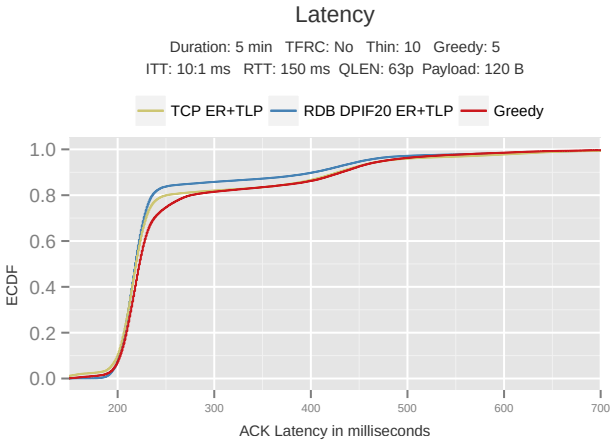
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	27 / 30 / 34	356/397/442	9/1/6/8/11/25
RDB DPIF20 ER+TLP	rdbsender	10	31 / 33 / 37	522/551/627	8/1/6/8/11/24
Greedy	zsender	5	22 / 24 / 26	1446/1447/1448	10/1/7/9/11/32

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
97.8 K	2985	2986	0	38.7 M	37.6 M	1.0 M	1.0 M	2.70 / 3.05	17595
90.2 K	3179	2114	36.5 K	49.5 M	37.1 M	12.4 M	1.1 M	3.55 / 3.52	13187
61.9 K	1871	1873	0	89.6 M	86.9 M	2.7 M	2.7 M	3.02 / 3.02	10936

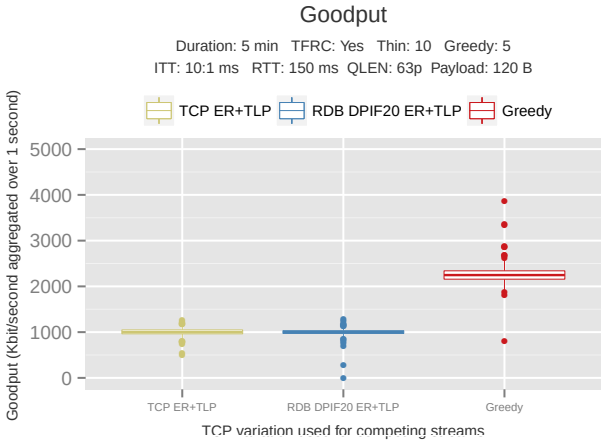
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
261	219	223	229	251	428	452	476	501	560	610	659	1132	1766	2558
252	218	222	227	236	403	433	454	468	492	570	647	1117	1618	2047
264	223	228	238	276	432	454	476	494	520	567	628	859	1151	1483

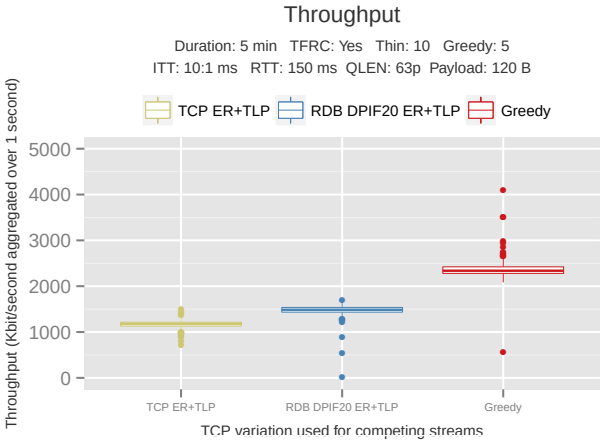
(c)

Figure A.3.29



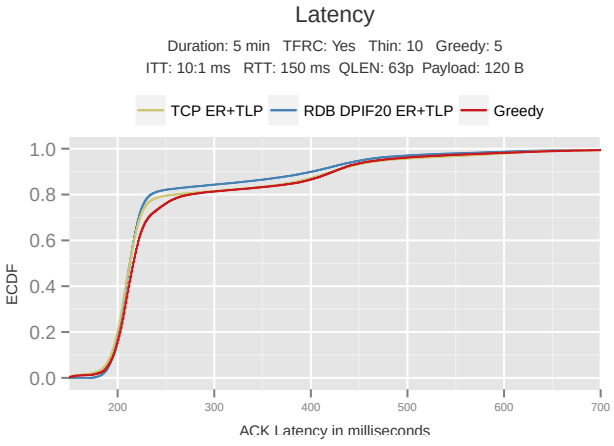
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	29 / 30 / 33	375/396/426	9/1/6/8/11/28
RDB DPIF20 ER+TLP	rdbsender	10	25 / 28 / 29	426/471/499	10/1/6/9/13/139
Greedy	zsender	5	22 / 24 / 26	1447/1447/1447	9/1/7/9/11/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
98.1 K	3221	3221	0	38.8 M	37.7 M	1.2 M	1.2 M	2.99 / 3.28	18092
105.7 K	3409	2399	34.7 K	49.7 M	37.1 M	12.6 M	851.0 K	3.14 / 3.22	17341
61.0 K	2078	2078	0	88.3 M	85.3 M	3.0 M	3.0 M	3.40 / 3.40	10876

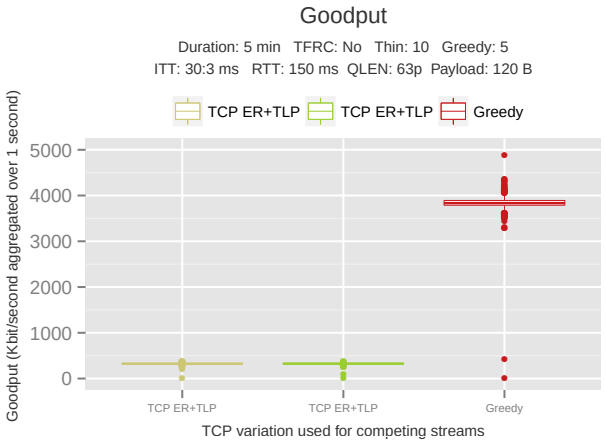
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
256	213	217	223	259	422	448	477	511	561	601	648	1121	1659	1936
249	213	217	222	234	401	430	453	469	499	553	623	1018	1446	1669
261	216	222	232	275	423	444	471	496	534	589	658	1632	3145	4561

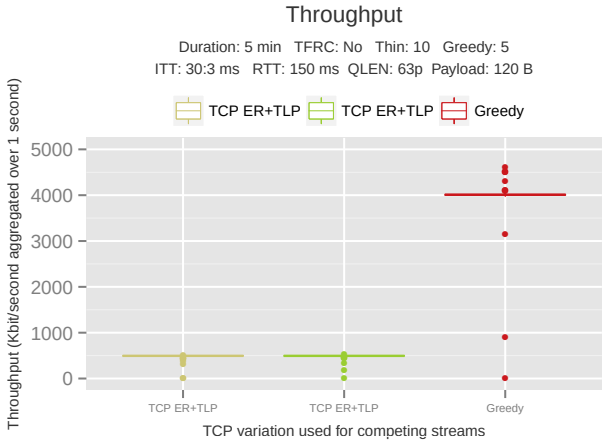
(c)

Figure A.3.30



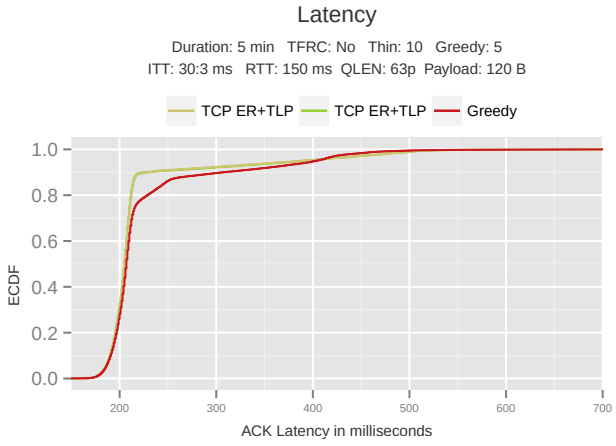
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	31 / 31 / 31	126/127/129	10/1/8/11/12/13
TCP ER+TLP	rdbsender	10	30 / 31 / 31	125/126/128	10/2/8/11/12/13
Greedy	zsender	5	14 / 15 / 16	1445/1447/1448	14/4/11/14/16/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
97.2 K	1204	1211	0	12.3 M	12.2 M	147.2 K	147.2 K	1.19 / 1.24	8633
96.1 K	1205	1212	0	12.2 M	12.0 M	147.7 K	147.7 K	1.21 / 1.25	8768
101.3 K	1323	1325	0	146.6 M	144.7 M	1.9 M	1.9 M	1.30 / 1.31	12507

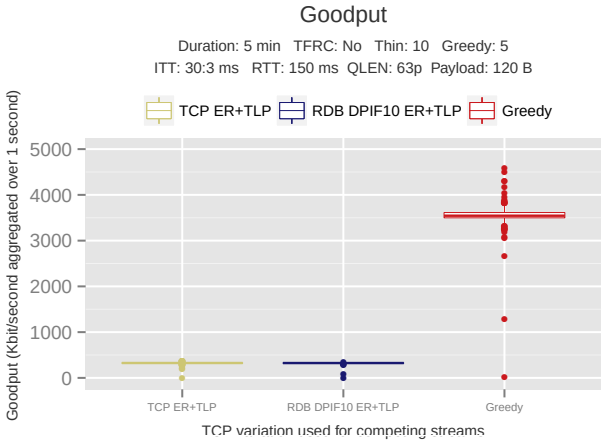
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
221	205	207	210	212	229	322	387	416	445	474	502	624	1528	1750
221	205	207	209	212	227	328	390	418	446	474	501	624	945	1152
228	207	210	213	229	307	372	404	413	422	441	468	640	746	1152

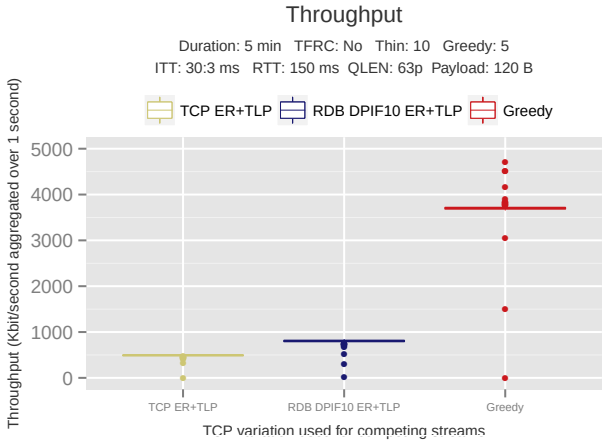
(c)

Figure A.3.31



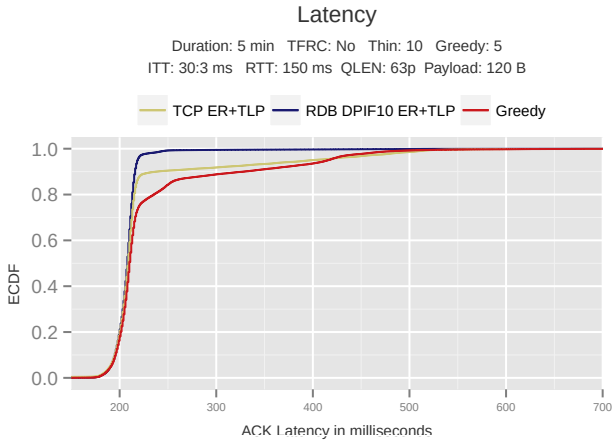
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	31 / 31 / 32	126/128/131	10/1/8/11/12/13
RDB DPIF10 ER+TLP	rdbsender	10	32 / 32 / 33	256/261/266	10/2/7/11/12/13
Greedy	zsender	5	15 / 16 / 17	1447/1447/1447	13/3/11/13/15/50

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
96.2 K	1233	1233	0	12.3 M	12.2 M	153.2 K	153.2 K	1.24 / 1.28	9018
91.8 K	1233	79	91.7 K	24.0 M	12.0 M	12.0 M	8.6 K	1.34 / 1.34	447
94.0 K	1369	1369	0	136.0 M	134.0 M	2.0 M	2.0 M	1.45 / 1.46	11917

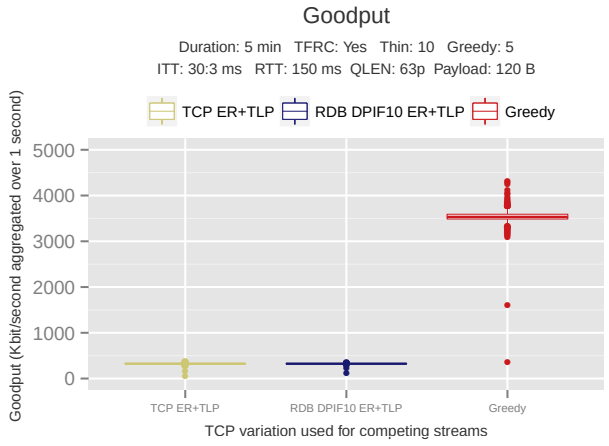
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
226	209	211	213	215	238	338	399	427	455	483	511	647	909	1154
209	208	210	211	213	216	217	219	220	222	230	245	512	780	1153
234	210	213	216	235	328	390	416	423	434	456	485	678	1058	1152

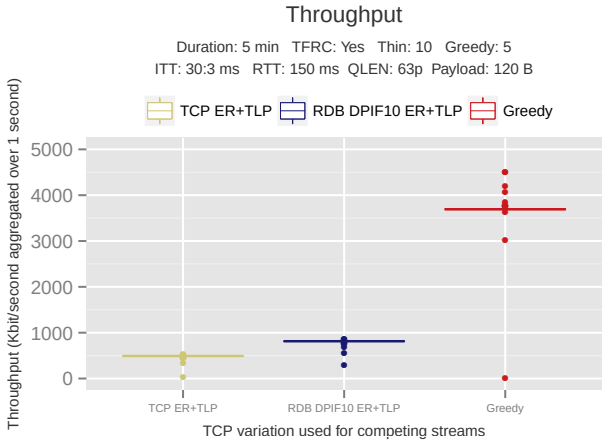
(c)

Figure A.3.32



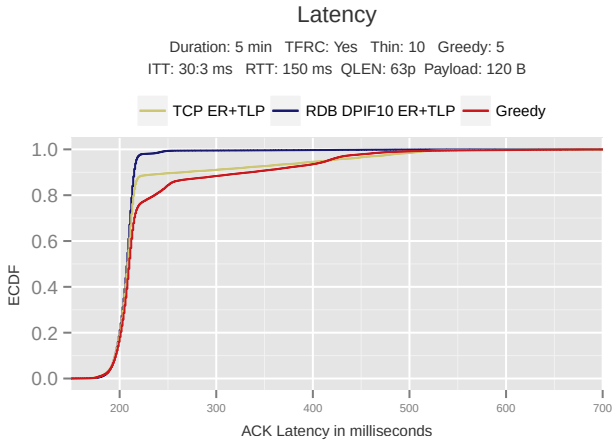
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	31 / 31 / 32	127/129/131	10/1/8/11/12/13
RDB DPIF10 ER+TLP	rdbsender	10	30 / 30 / 30	241/243/245	11/1/10/12/12/25
Greedy	zsender	5	15 / 16 / 17	1446/1447/1447	13/4/10/13/16/36

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
95.8 K	1357	1357	0	12.4 M	12.2 M	167.4 K	167.4 K	1.35 / 1.42	9723
99.0 K	1406	72	98.9 K	24.1 M	12.0 M	12.0 M	10.1 K	1.41 / 1.42	492
93.5 K	1388	1388	0	135.3 M	133.3 M	2.0 M	2.0 M	1.48 / 1.48	12396

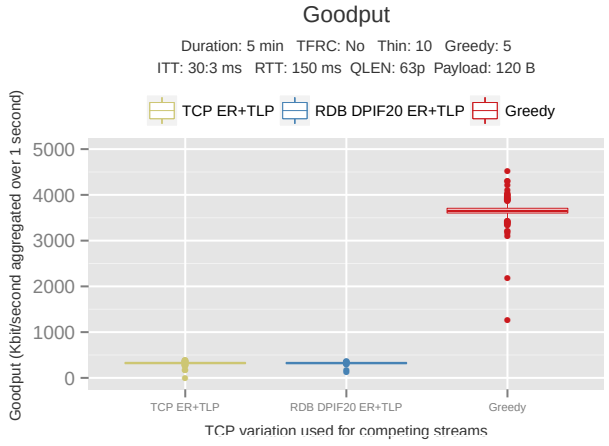
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
227	208	210	212	215	265	358	414	438	462	486	511	678	852	1049
208	208	209	211	213	215	216	217	218	219	225	244	503	779	1242
235	210	212	215	236	335	392	415	422	432	455	489	671	866	999

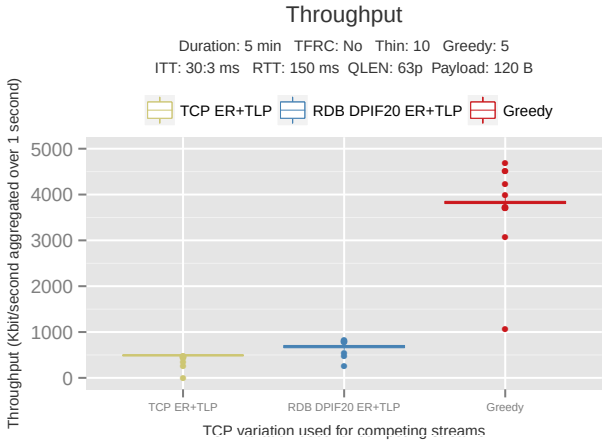
(c)

Figure A.3.33



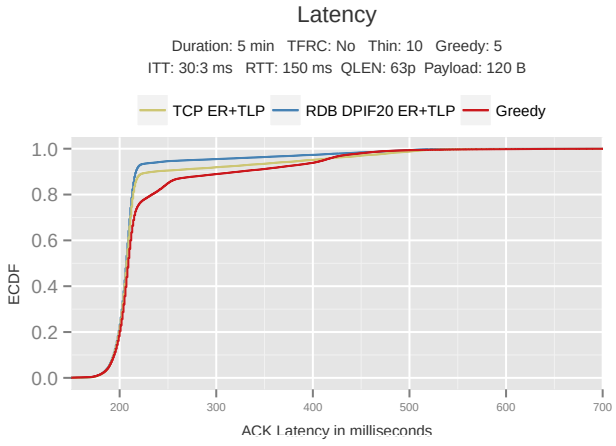
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	31 / 31 / 31	126/128/130	10/1/8/11/12/13
RDB DPIF20 ER+TLP	rdbsender	10	31 / 31 / 32	201/205/211	10/2/8/11/12/13
Greedy	zsender	5	15 / 15 / 16	1446/1447/1448	14/4/11/13/16/46

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
96.3 K	1238	1238	0	12.3 M	12.2 M	152.8 K	152.8 K	1.24 / 1.28	9014
94.9 K	1252	680	56.5 K	19.5 M	12.0 M	7.4 M	82.2 K	1.24 / 1.32	5058
96.7 K	1411	1411	0	140.0 M	138.0 M	2.0 M	2.0 M	1.46 / 1.46	12464

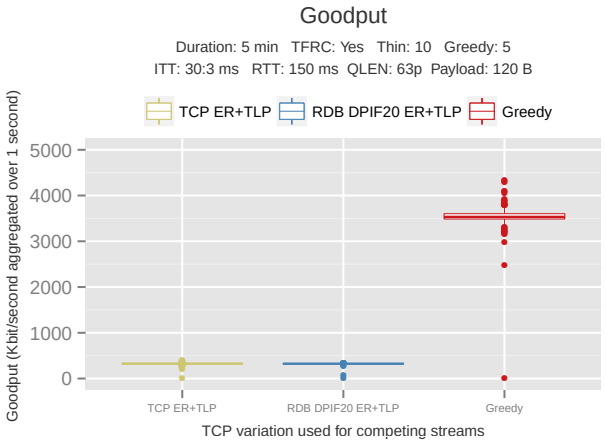
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
224	207	209	211	214	236	335	395	422	450	478	506	603	941	1276
216	207	209	211	213	217	222	272	329	384	433	481	564	835	999
232	209	212	215	233	324	386	412	420	430	450	477	640	808	999

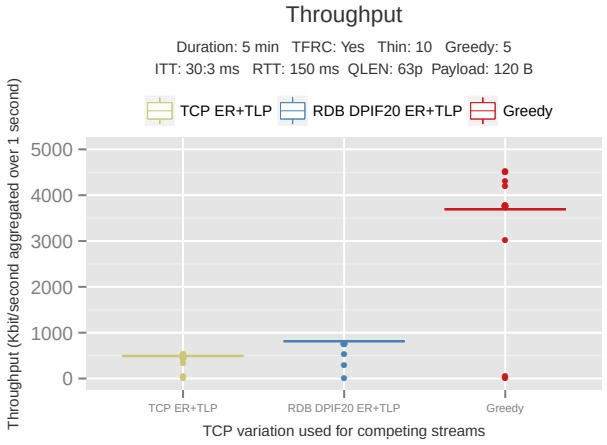
(c)

Figure A.3.34



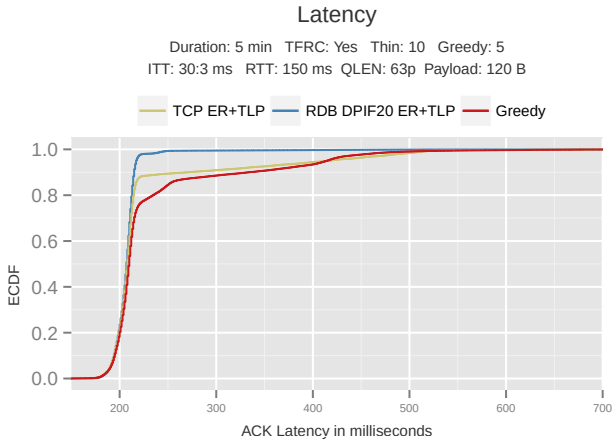
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	31 / 31 / 32	127/129/131	10/1/8/11/12/13
RDB DPIF20 ER+TLP	rdbsender	10	30 / 30 / 30	242/243/244	11/1/10/12/12/55
Greedy	zsender	5	15 / 16 / 16	1446/1447/1448	13/1/10/13/15/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
95.8 K	1395	1402	0	12.4 M	12.2 M	173.3 K	173.3 K	1.40 / 1.46	9834
98.5 K	1375	82	98.2 K	24.0 M	12.0 M	12.0 M	9.0 K	1.39 / 1.39	506
93.6 K	1413	1415	0	135.5 M	133.4 M	2.0 M	2.0 M	1.51 / 1.51	12145

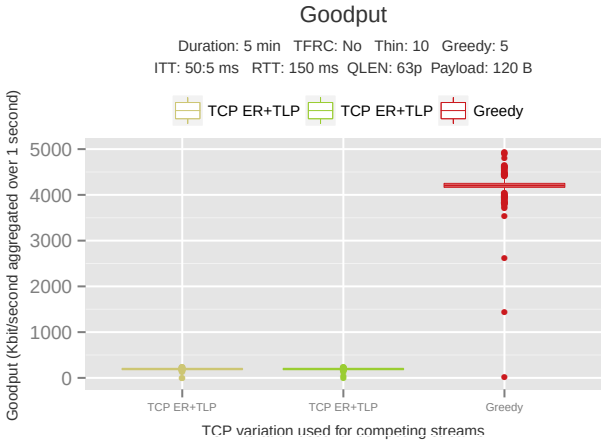
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
227	208	210	212	215	270	361	416	441	466	488	513	704	1065	1627
208	207	209	211	213	215	216	217	218	220	225	244	507	948	1288
235	210	212	215	235	334	393	414	422	433	457	494	678	2139	2321

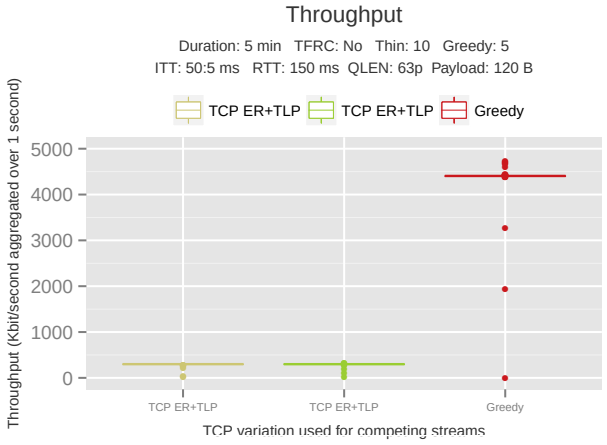
(c)

Figure A.3.35



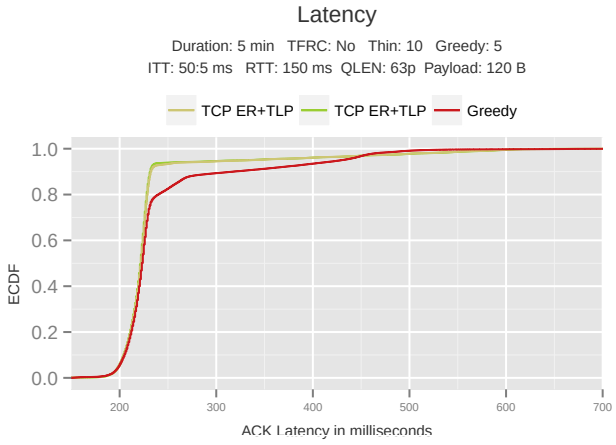
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 50 / 50	121/122/123	8/1/8/9/9/10
TCP ER+TLP	rdbsender	10	50 / 50 / 50	122/122/123	8/1/8/9/9/10
Greedy	zsender	5	13 / 13 / 14	1447/1447/1448	17/3/13/16/19/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
60.2 K	589	596	0	7.3 M	7.3 M	70.8 K	70.8 K	0.96 / 0.98	3410
59.3 K	588	596	0	7.2 M	7.2 M	70.9 K	70.9 K	0.98 / 0.99	3254
110.9 K	1198	1200	0	160.5 M	158.7 M	1.7 M	1.7 M	1.08 / 1.08	13251

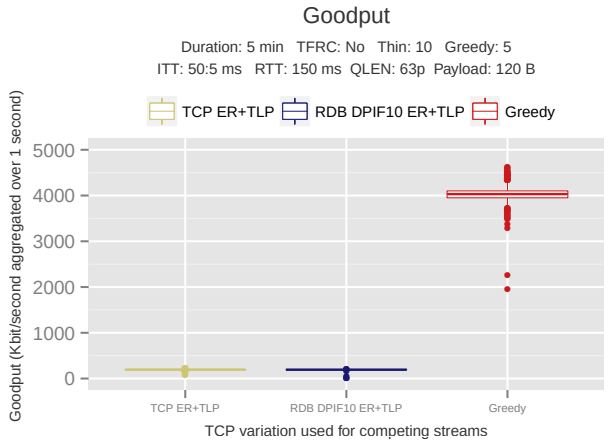
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
233	222	224	226	229	232	243	336	395	451	509	567	640	1152	1152
233	222	224	226	228	232	235	331	395	452	514	573	652	1151	1152
244	223	226	229	240	318	390	430	443	452	465	495	668	888	1151

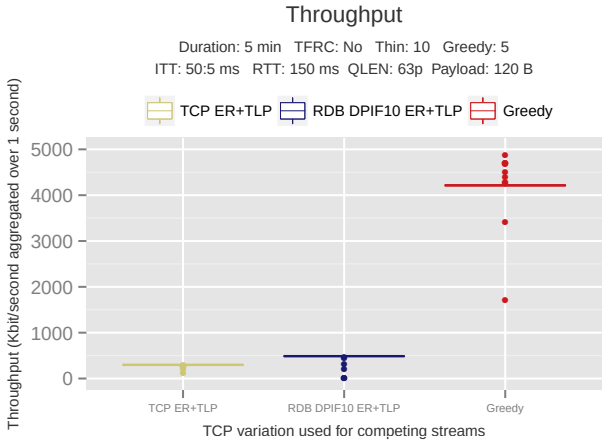
(c)

Figure A.3.36



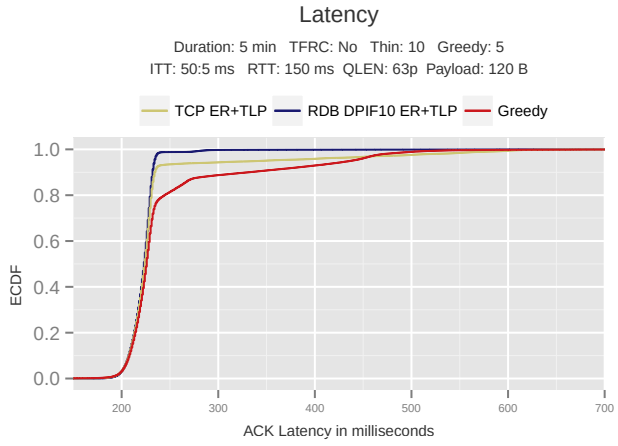
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 50 / 50	122/122/123	8/1/8/9/9/10
RDB DPIF10 ER+TLP	rdbsender	10	51 / 51 / 52	242/244/248	8/1/8/9/9/10
Greedy	zsender	5	13 / 14 / 15	1447/1447/1447	16/6/13/15/18/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
60.0 K	612	612	0	7.3 M	7.3 M	74.0 K	74.0 K	1.01 / 1.02	3431
58.3 K	567	30	58.3 K	14.2 M	7.1 M	7.1 M	2.5 K	0.96 / 0.97	118
106.3 K	1189	1189	0	153.9 M	152.2 M	1.7 M	1.7 M	1.12 / 1.12	13294

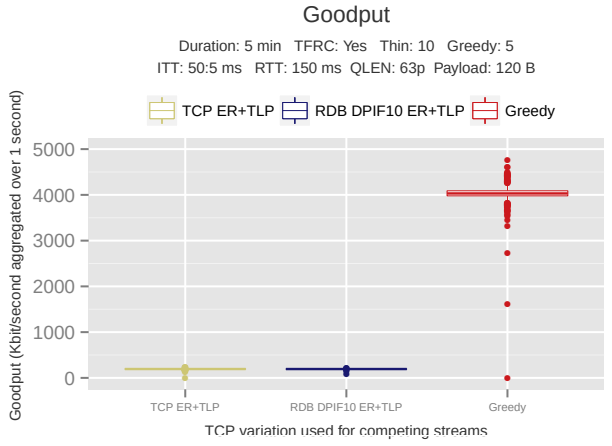
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
236	224	226	228	231	234	240	346	405	461	522	578	661	813	1086
223	223	226	228	230	232	233	234	235	236	237	273	510	1252	3246
248	226	229	232	244	331	401	437	449	458	472	502	673	943	999

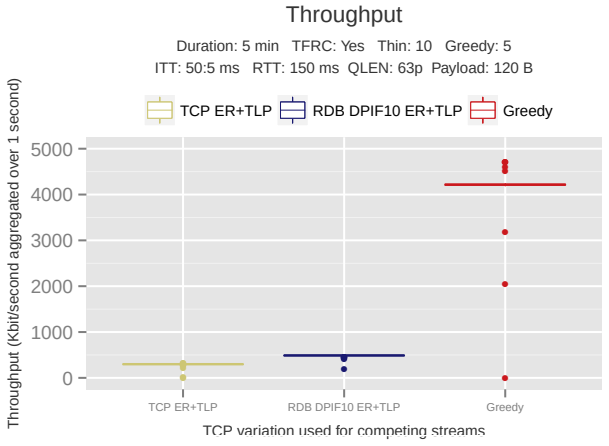
(c)

Figure A.3.37



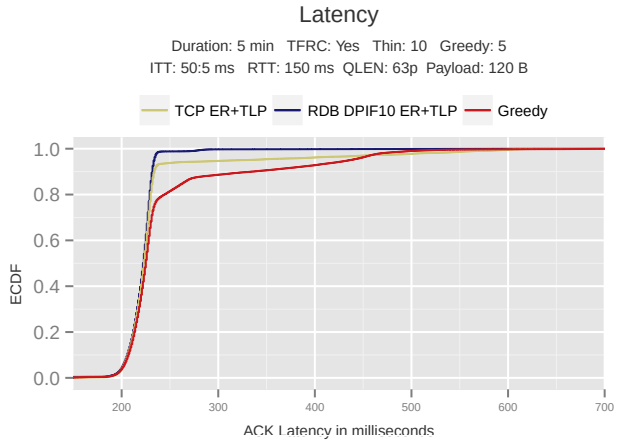
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 50 / 50	121/122/123	8/1/8/9/9/10
RDB DPIF10 ER+TLP	rdbsender	10	49 / 50 / 50	240/240/241	10/1/9/10/11/22
Greedy	zsender	5	13 / 14 / 15	1447/1447/1447	16/1/13/15/19/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
60.2 K	568	575	0	7.3 M	7.3 M	68.3 K	68.3 K	0.93 / 0.94	3252
59.8 K	570	28	59.8 K	14.4 M	7.2 M	7.2 M	3.4 K	0.95 / 0.95	139
106.2 K	1224	1226	0	153.8 M	152.0 M	1.8 M	1.8 M	1.15 / 1.15	13317

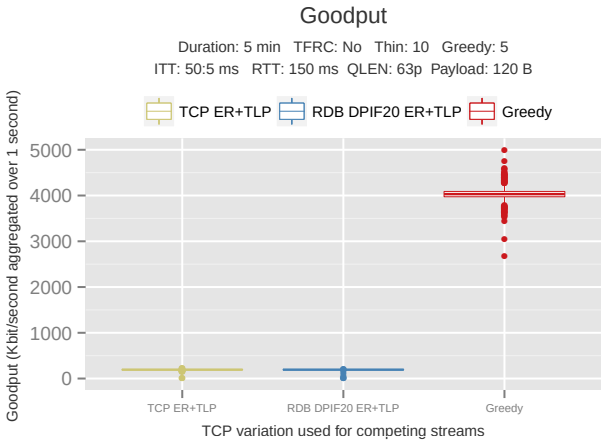
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
235	223	226	228	230	234	237	322	391	450	508	567	632	1150	1151
222	222	225	227	229	232	233	234	234	235	236	274	524	751	963
247	225	228	231	244	334	403	436	448	456	469	499	686	2201	2330

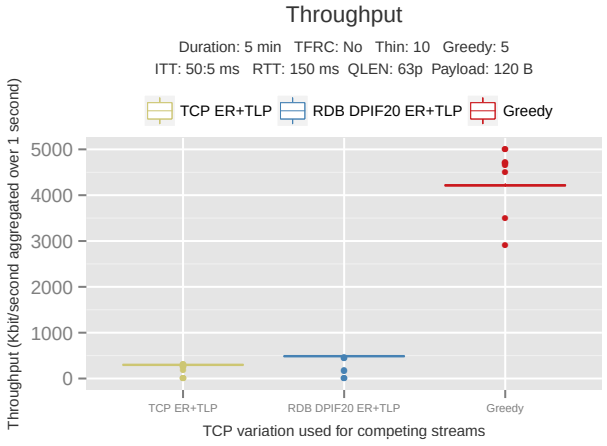
(c)

Figure A.3.38



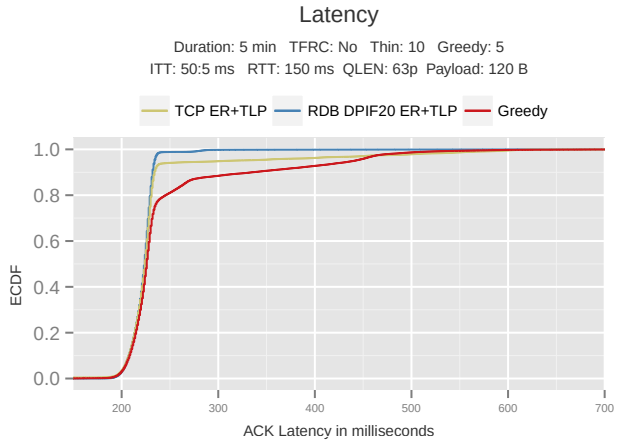
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 50 / 51	121/122/122	8/1/8/9/9/10
RDB DPIF20 ER+TLP	rdbsender	10	51 / 51 / 52	244/245/248	8/1/8/9/9/10
Greedy	zsender	5	13 / 14 / 15	1447/1447/1447	16/6/13/16/19/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
60.0 K	542	550	0	7.3 M	7.3 M	64.7 K	64.7 K	0.88 / 0.90	3171
58.1 K	577	31	58.0 K	14.2 M	7.1 M	7.1 M	2.0 K	0.99 / 0.99	92
106.5 K	1227	1227	0	154.2 M	152.4 M	1.8 M	1.8 M	1.15 / 1.15	13299

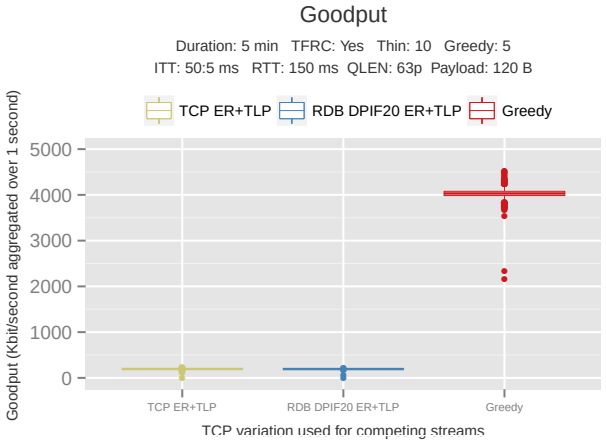
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
235	224	226	228	231	234	237	310	383	448	505	565	647	1301	3268
223	224	226	228	230	232	233	234	235	236	237	273	465	3171	3412
249	226	229	232	246	335	405	441	451	460	477	516	671	787	999

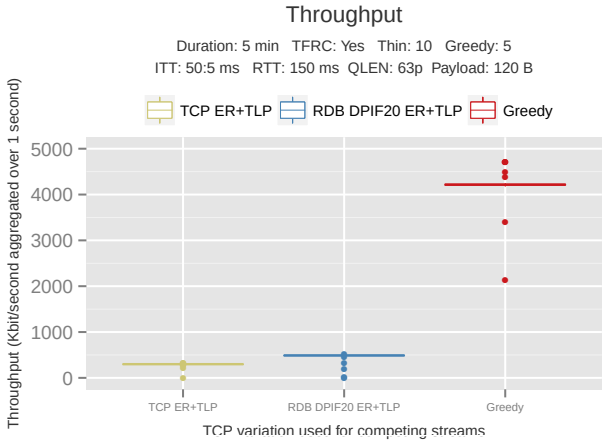
(c)

Figure A.3.39



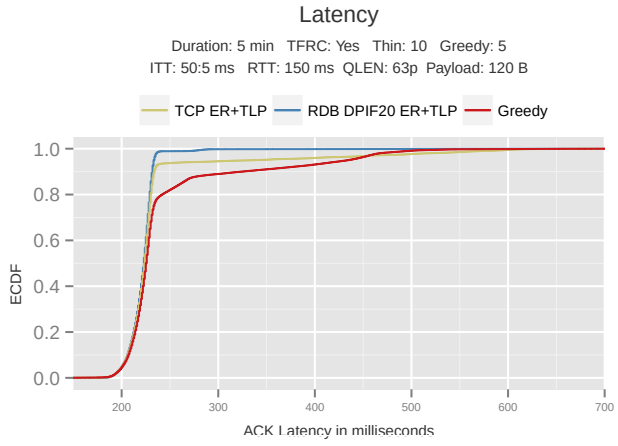
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	50 / 50 / 50	121/122/123	8/1/8/9/9/11
RDB DPIF20 ER+TLP	rdbsender	10	50 / 50 / 50	240/240/241	12/2/9/10/12/78
Greedy	zsender	5	13 / 14 / 15	1447/1447/1447	16/5/13/15/19/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
60.2 K	603	603	0	7.4 M	7.3 M	72.4 K	72.4 K	0.98 / 1.00	3316
59.6 K	538	31	59.5 K	14.3 M	7.2 M	7.2 M	2.8 K	0.90 / 0.90	101
106.2 K	1199	1199	0	153.8 M	152.0 M	1.7 M	1.7 M	1.13 / 1.13	12920

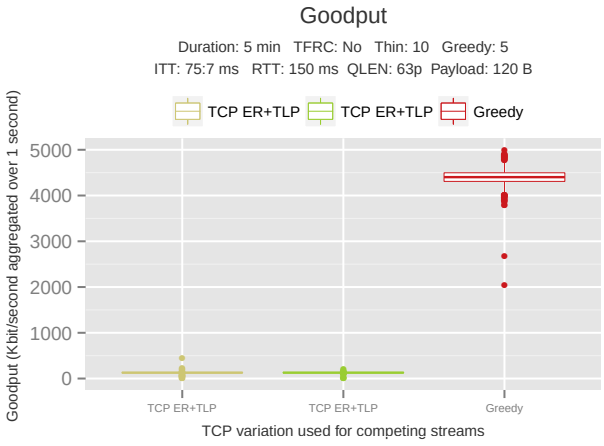
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
235	224	226	228	230	234	237	338	401	457	516	574	644	723	869
222	223	225	227	229	232	233	233	234	235	236	271	508	1243	1244
246	225	228	231	242	324	398	434	446	456	467	496	635	735	999

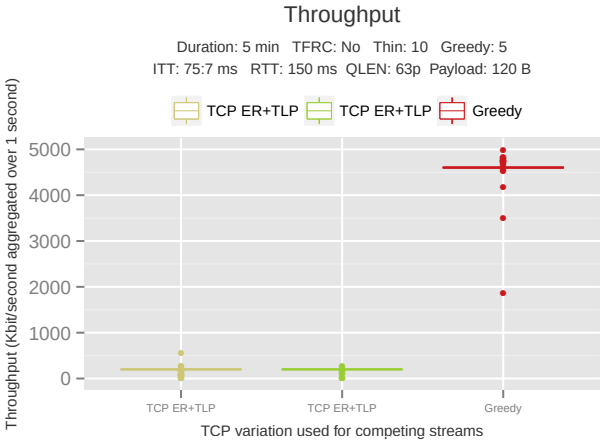
(c)

Figure A.3.40



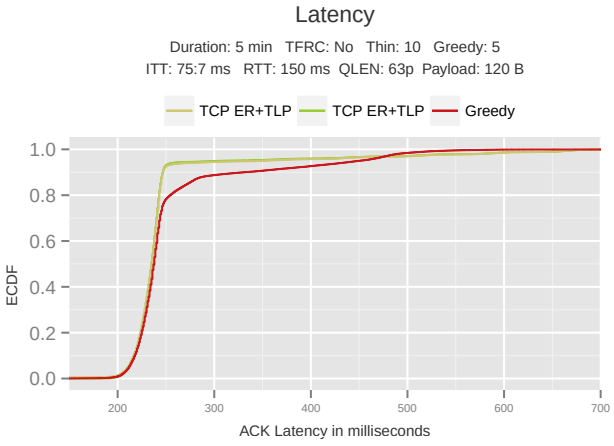
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	75 / 76 / 78	120/123/127	7/1/7/8/8/10
TCP ER+TLP	rdbsender	10	74 / 75 / 77	120/121/124	7/1/7/8/8/10
Greedy	zsender	5	12 / 13 / 13	1447/1447/1447	19/5/15/18/22/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
39.8 K	455	463	0	4.9 M	4.8 M	58.2 K	58.2 K	1.19 / 1.14	2059
39.6 K	424	433	0	4.8 M	4.7 M	52.1 K	52.1 K	1.08 / 1.07	1891
115.8 K	1102	1102	0	167.5 M	165.9 M	1.6 M	1.6 M	0.95 / 0.95	13786

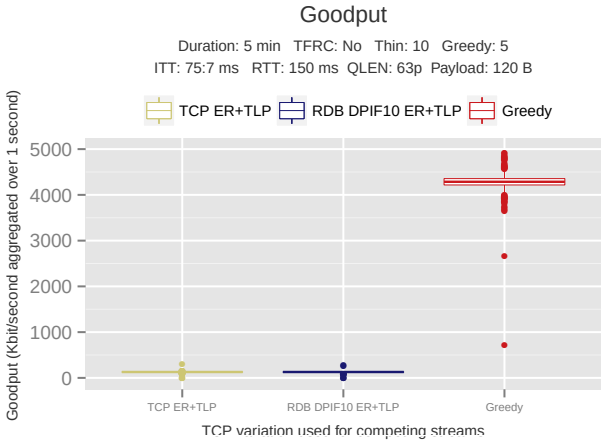
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
250	235	238	241	243	247	252	350	428	502	577	651	746	7752	7779
248	235	238	240	243	247	250	307	397	487	572	648	769	1455	7766
258	238	241	244	254	331	407	450	466	477	490	518	603	775	999

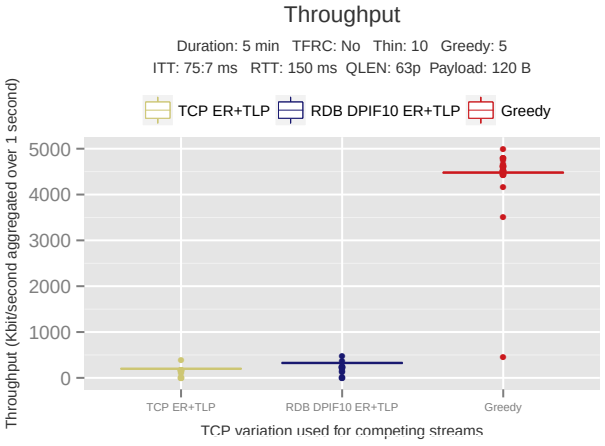
(c)

Figure A.3.41



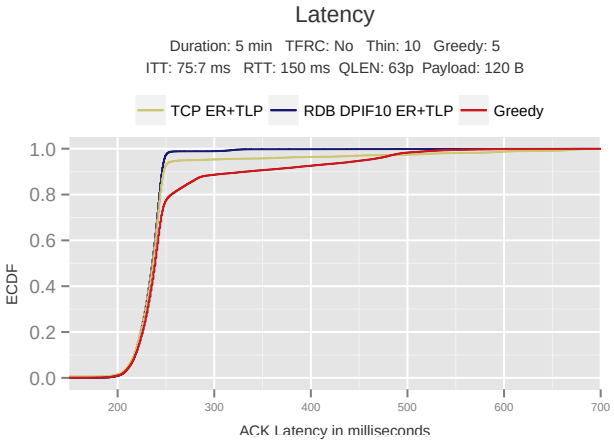
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	75 / 75 / 77	120/121/123	7/1/8/8/8/10
RDB DPIF10 ER+TLP	rdbsender	10	75 / 76 / 78	240/241/244	7/1/8/8/8/10
Greedy	zsender	5	12 / 13 / 14	1446/1447/1447	18/5/15/17/21/46

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.2 K	388	395	0	4.9 M	4.8 M	46.0 K	46.0 K	0.94 / 0.96	1754
39.0 K	367	30	39.0 K	9.4 M	4.7 M	4.7 M	7.6 K	0.97 / 0.94	56
112.6 K	1082	1082	0	162.9 M	161.4 M	1.6 M	1.6 M	0.96 / 0.96	13468

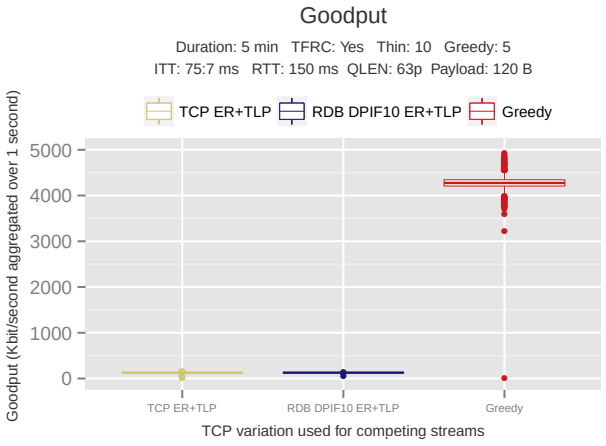
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
247	237	239	242	244	248	250	277	365	451	536	632	705	3262	7748
236	236	239	241	243	246	247	248	249	250	251	305	599	3262	7747
260	239	242	245	256	333	411	456	471	482	494	524	670	820	999

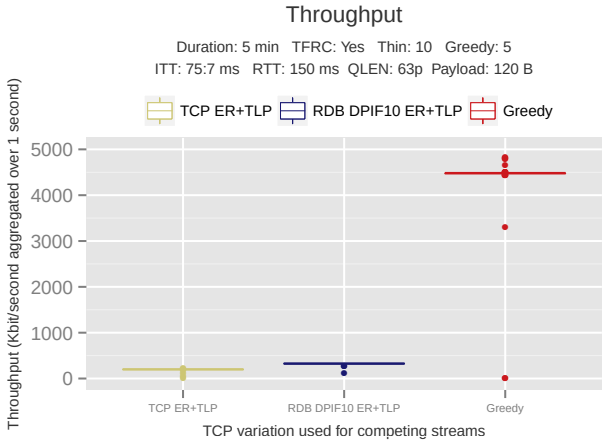
(c)

Figure A.3.42



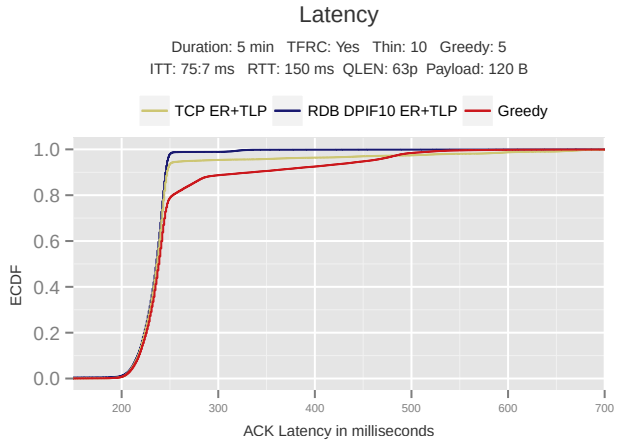
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	74 / 74 / 74	120/120/120	7/1/8/8/8/10
RDB DPIF10 ER+TLP	rdbsender	10	74 / 75 / 75	240/240/240	10/1/8/8/10/23
Greedy	zsender	5	11 / 13 / 15	1446/1446/1447	18/6/15/17/21/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.5 K	394	403	0	4.9 M	4.8 M	47.5 K	47.5 K	0.97 / 0.97	1760
39.8 K	399	19	39.8 K	9.6 M	4.8 M	4.8 M	2.3 K	1.00 / 1.00	65
112.5 K	1133	1135	0	162.8 M	161.1 M	1.6 M	1.6 M	1.01 / 1.01	13394

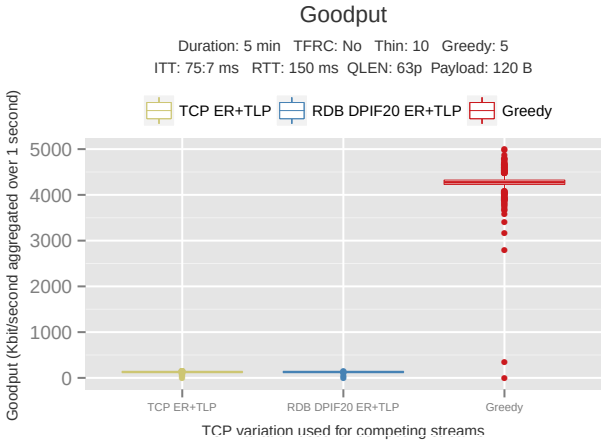
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
247	237	239	242	244	247	249	275	363	452	540	642	746	1151	1153
235	236	239	241	243	246	247	247	248	249	250	308	527	746	907
259	239	242	245	253	335	412	455	470	481	492	519	670	937	1151

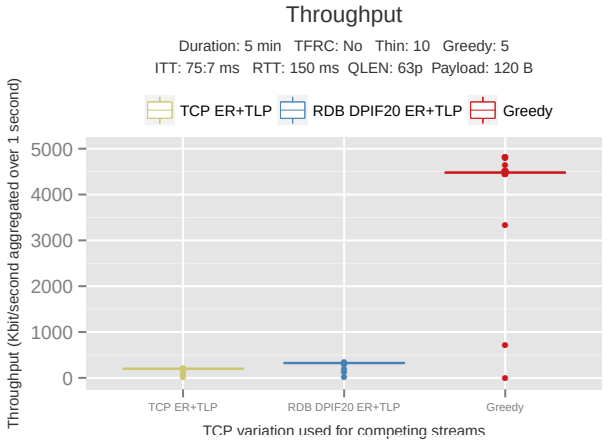
(c)

Figure A.3.43



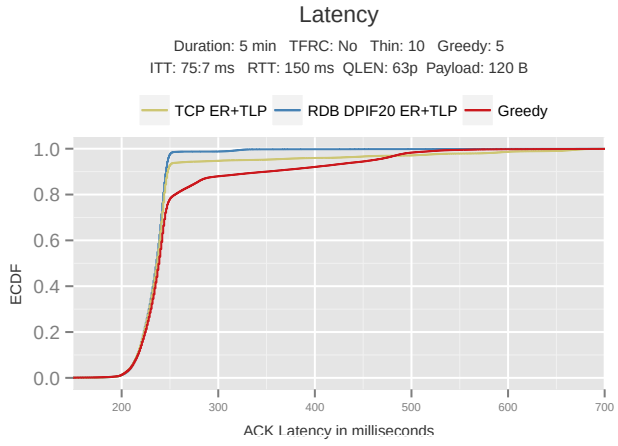
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	74 / 74 / 74	120/120/120	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	10	75 / 75 / 75	240/241/241	7/1/7/8/8/10
Greedy	zsender	5	12 / 13 / 14	1447/1447/1447	19/3/14/18/22/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.6 K	438	446	0	4.9 M	4.8 M	52.7 K	52.7 K	1.08 / 1.08	2002
39.5 K	423	27	39.5 K	9.5 M	4.8 M	4.8 M	2.4 K	1.08 / 1.07	80
112.6 K	1199	1201	0	163.0 M	161.2 M	1.7 M	1.7 M	1.07 / 1.06	14271

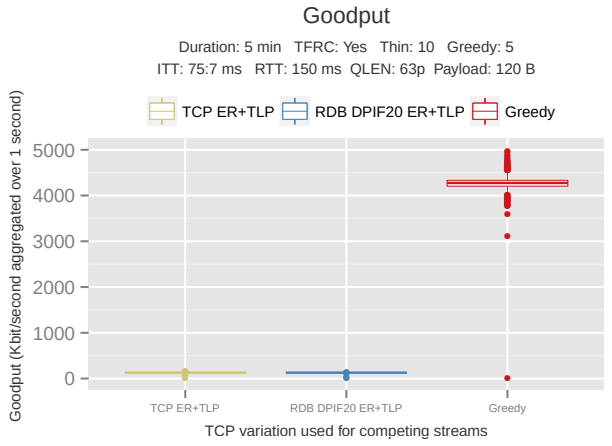
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
248	237	240	242	244	248	250	322	411	491	573	645	687	1151	1151
235	236	239	241	243	246	247	248	248	249	251	312	595	1152	1153
260	239	242	245	255	350	420	460	474	483	494	522	700	792	1152

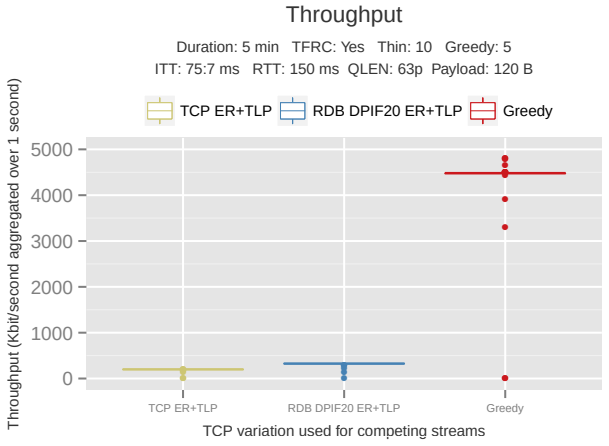
(c)

Figure A.3.44



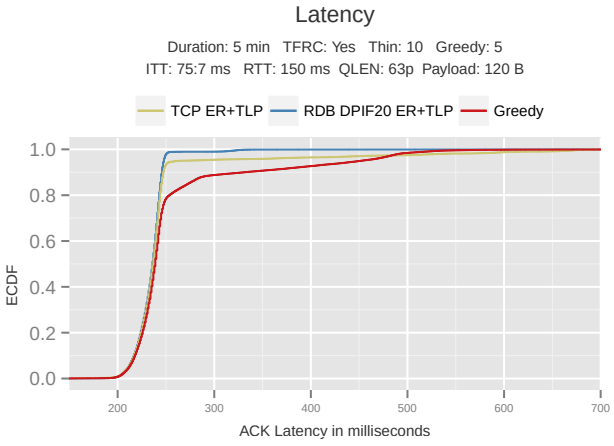
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	74 / 74 / 74	120/120/120	7/1/8/8/8/10
RDB DPIF20 ER+TLP	rdbsender	10	75 / 75 / 75	240/240/240	12/1/8/9/12/91
Greedy	zsender	5	13 / 13 / 14	1446/1447/1447	18/4/14/17/21/47

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.6 K	379	387	0	4.9 M	4.8 M	45.6 K	45.6 K	0.93 / 0.93	1715
39.7 K	377	14	39.7 K	9.5 M	4.8 M	4.8 M	840	0.95 / 0.95	30
112.4 K	1089	1091	0	162.6 M	161.0 M	1.6 M	1.6 M	0.97 / 0.97	13364

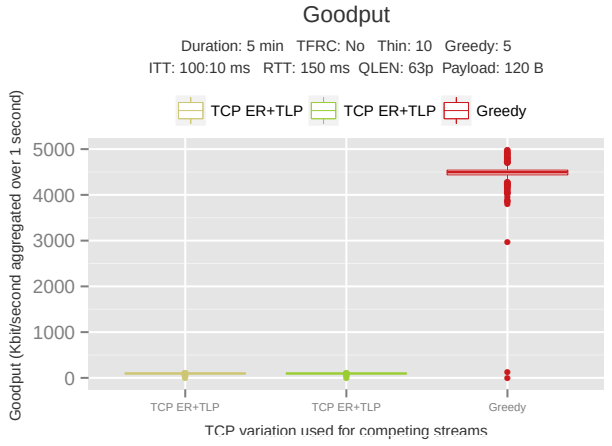
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
247	237	240	242	244	247	250	264	360	447	534	622	704	1152	1152
235	236	239	241	243	246	247	248	248	249	250	302	368	1151	1153
259	239	242	245	254	331	407	452	469	480	491	519	658	806	1152

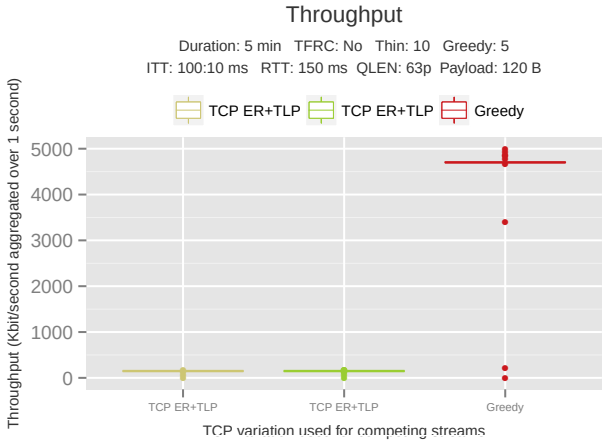
(c)

Figure A.3.45



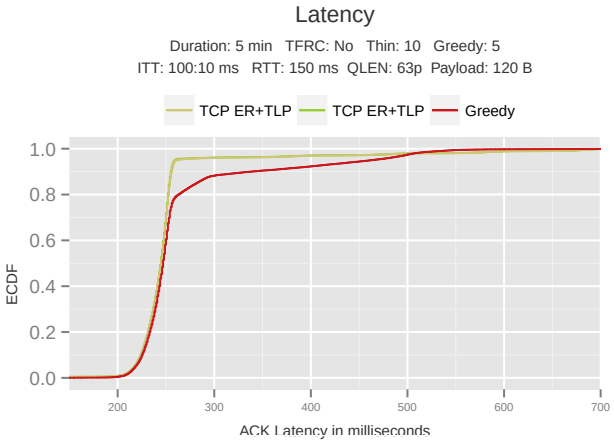
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	98 / 99 / 99	120/120/120	7/1/7/7/7/10
TCP ER+TLP	rdbsender	10	98 / 99 / 99	120/120/120	7/1/7/7/7/10
Greedy	zsender	5	12 / 13 / 13	1446/1447/1447	20/6/16/19/23/46

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.4 K	305	311	0	3.7 M	3.6 M	37.2 K	37.2 K	1.00 / 1.00	1125
30.1 K	292	299	0	3.6 M	3.6 M	35.5 K	35.5 K	0.96 / 0.97	1071
118.0 K	1084	1084	0	170.7 M	169.2 M	1.6 M	1.6 M	0.92 / 0.92	14060

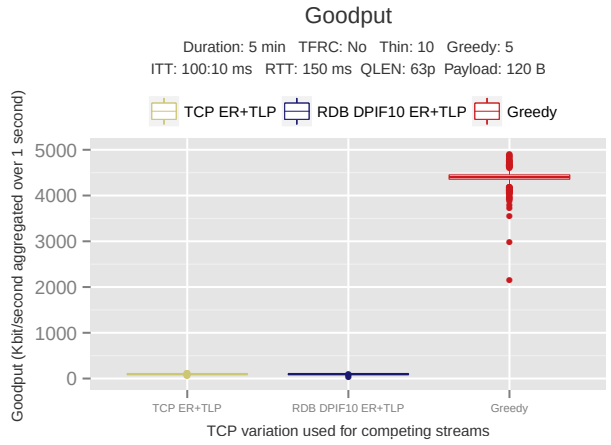
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
253	245	248	250	253	256	257	261	304	408	534	650	703	830	965
253	245	248	250	252	255	257	259	291	399	508	651	758	930	1030
267	247	250	254	263	338	416	463	482	496	508	534	708	831	1256

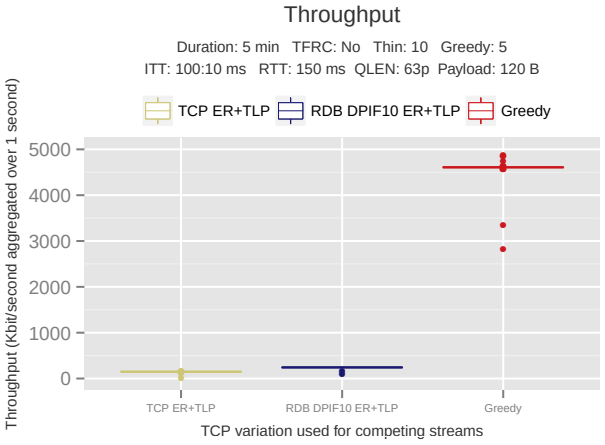
(c)

Figure A.3.46



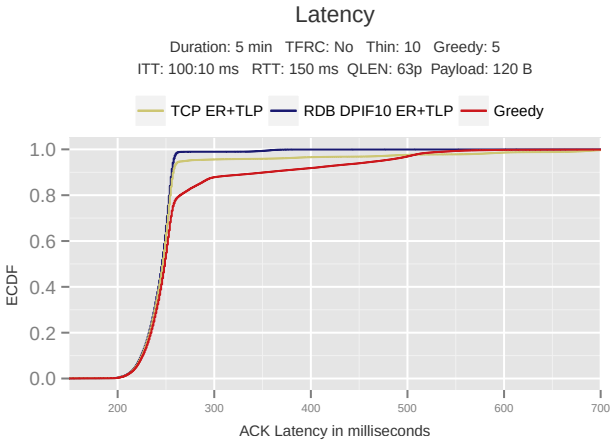
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	98 / 99 / 99	120/120/120	6/1/7/7/7/10
RDB DPIF10 ER+TLP	rdbsender	10	99 / 100 / 100	240/240/240	7/1/7/7/7/10
Greedy	zsender	5	11 / 13 / 13	1446/1447/1447	20/7/15/19/23/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.5 K	332	343	0	3.7 M	3.6 M	41.2 K	41.2 K	1.09 / 1.09	1228
29.8 K	294	10	29.8 K	7.2 M	3.6 M	3.6 M	1.2 K	0.98 / 0.99	26
115.9 K	1096	1096	0	167.7 M	166.1 M	1.6 M	1.6 M	0.95 / 0.95	14166

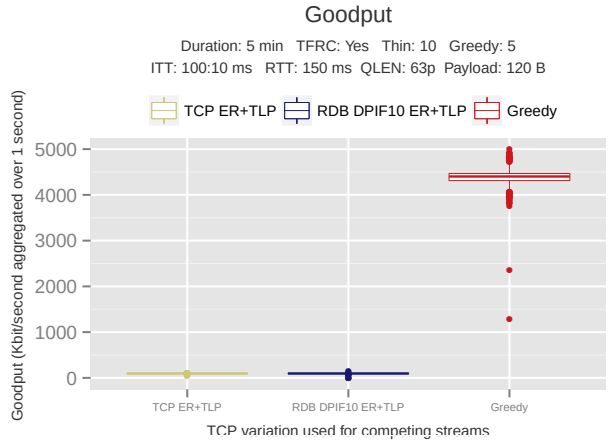
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
256	247	250	252	255	258	260	271	362	468	569	670	773	1007	1147
244	246	249	251	254	256	257	258	258	259	260	333	374	752	814
270	249	252	255	265	353	427	472	488	501	512	538	707	827	999

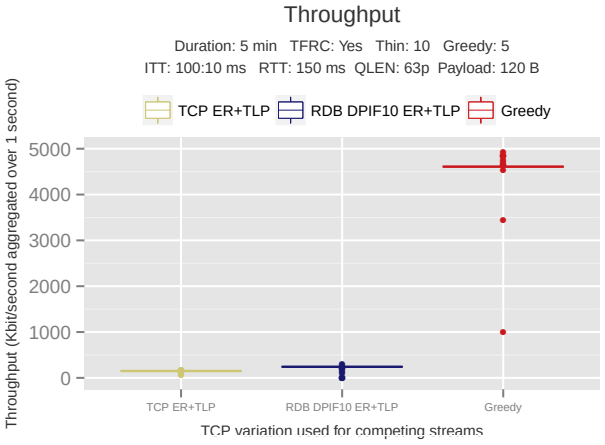
(c)

Figure A.3.47



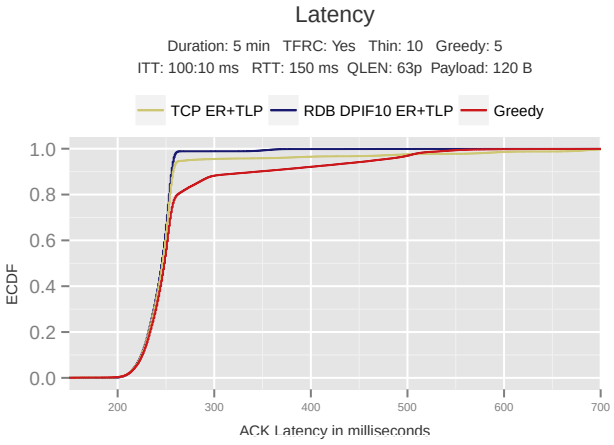
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	98 / 99 / 99	120/120/120	6/1/7/7/7/10
RDB DPIF10 ER+TLP	rdbsender	10	100 / 101 / 103	240/240/242	10/1/7/9/11/30
Greedy	zsender	5	12 / 13 / 13	1446/1447/1447	20/6/16/19/23/4

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.5 K	339	355	0	3.7 M	3.6 M	42.6 K	42.6 K	1.11 / 1.11	1247
29.4 K	303	31	29.4 K	7.1 M	3.5 M	3.5 M	1.7 K	0.99 / 1.03	30
115.8 K	1074	1074	0	167.6 M	166.0 M	1.6 M	1.6 M	0.93 / 0.93	13920

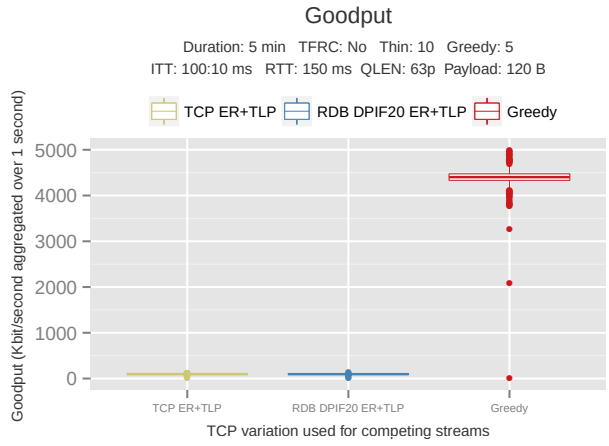
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
256	246	250	252	254	258	259	273	367	471	572	673	783	941	1033
245	246	249	251	253	256	257	258	258	259	260	340	576	3247	7707
269	248	252	255	262	345	423	468	487	501	511	538	690	803	999

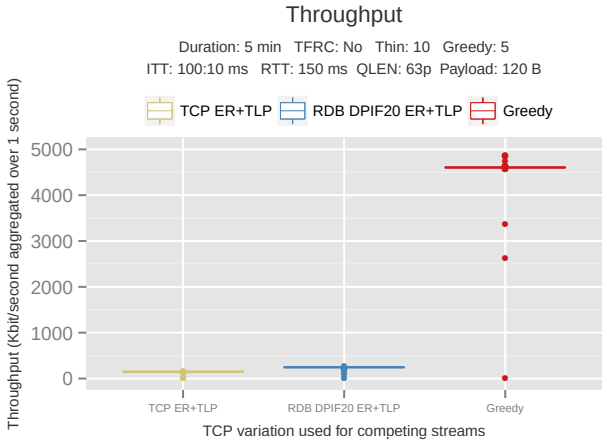
(c)

Figure A.3.48



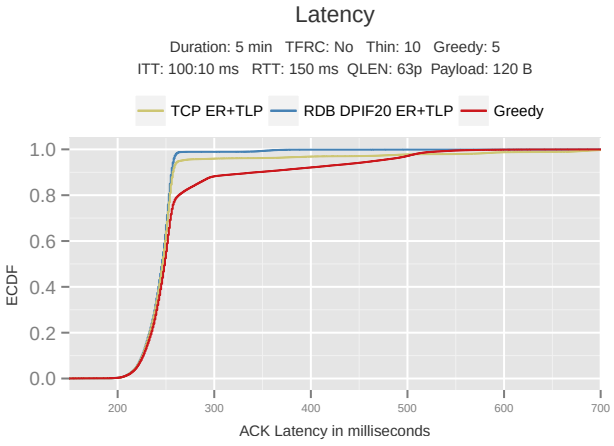
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	98 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF20 ER+TLP	rdbsender	10	100 / 100 / 102	240/240/242	7/1/7/7/7/10
Greedy	zsender	5	12 / 13 / 14	1446/1447/1447	19/7/16/19/22/4

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.5 K	296	313	0	3.7 M	3.6 M	36.7 K	36.7 K	0.97 / 0.97	1111
29.7 K	288	24	29.7 K	7.1 M	3.6 M	3.6 M	3.4 K	0.98 / 0.97	38
115.8 K	1054	1055	0	167.6 M	166.1 M	1.5 M	1.5 M	0.91 / 0.91	13814

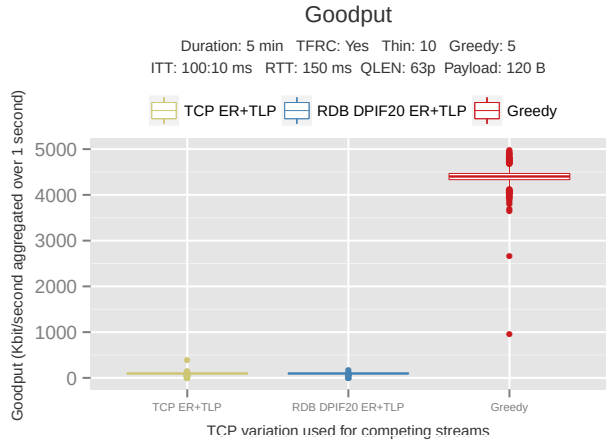
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
256	247	249	252	254	257	259	265	303	418	557	669	788	1150	1152
245	246	249	251	253	256	257	258	258	259	261	335	569	1214	5733
269	249	252	255	263	343	422	467	486	499	509	528	614	742	1151

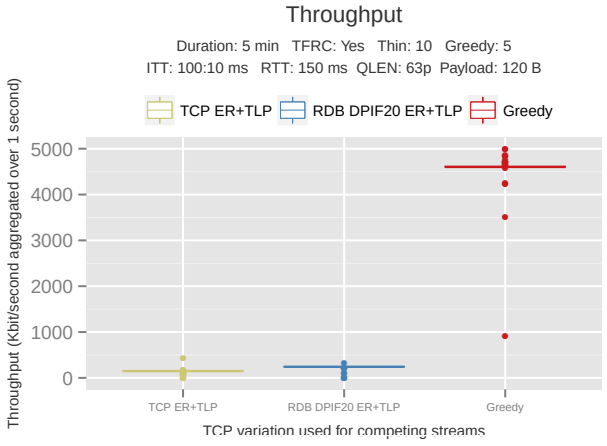
(c)

Figure A.3.49



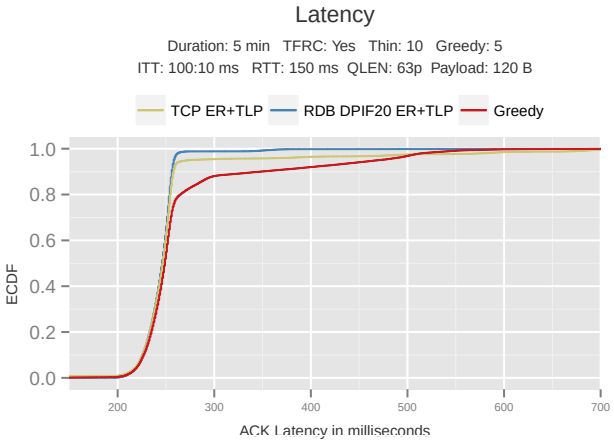
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	99 / 101 / 102	120/122/123	6/1/7/7/7/10
RDB DPIF20 ER+TLP	rdbsender	10	100 / 101 / 103	240/240/243	14/1/7/9/14/120
Greedy	zsender	5	12 / 13 / 13	1447/1447/1447	20/7/15/19/23/4

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.0 K	344	361	0	3.7 M	3.6 M	41.6 K	41.6 K	1.10 / 1.14	1215
29.4 K	302	33	29.4 K	7.1 M	3.5 M	3.5 M	2.2 K	0.99 / 1.02	47
115.7 K	1035	1035	0	167.5 M	166.0 M	1.5 M	1.5 M	0.89 / 0.89	13973

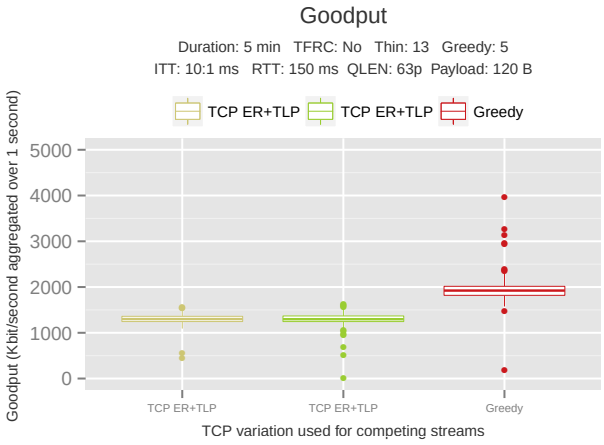
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
258	247	250	252	255	258	260	274	376	475	578	676	837	7764	7787
246	246	249	251	254	256	257	258	259	260	262	342	638	3266	7777
270	249	252	255	265	348	425	470	489	501	515	546	703	812	999

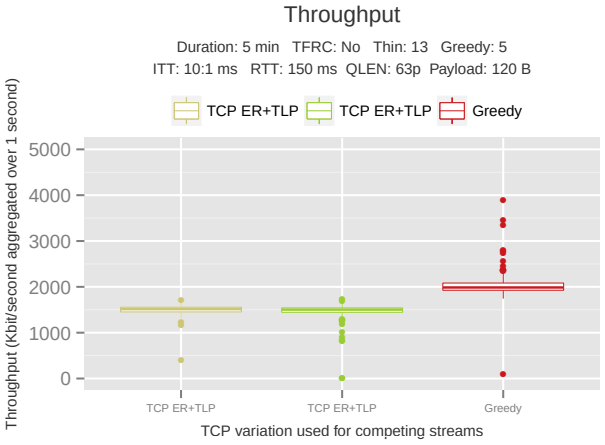
(c)

Figure A.3.50



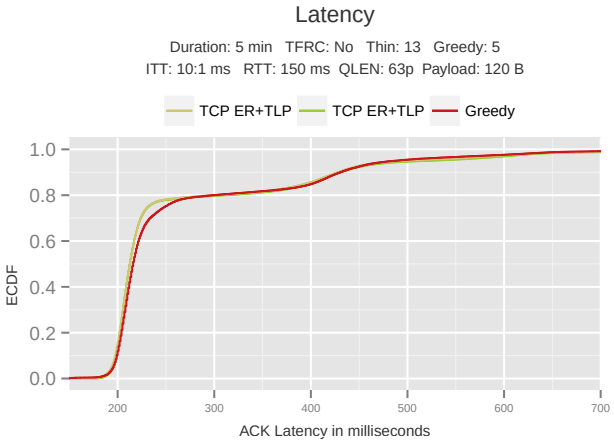
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	31 / 33 / 36	399/433/472	8/1/5/7/10/27
TCP ER+TLP	rdbsender	13	31 / 35 / 38	407/454/501	8/1/5/7/10/23
Greedy	zsender	5	27 / 28 / 31	1447/1447/1447	8/1/6/8/10/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
116.8 K	4192	4192	0	50.5 M	48.9 M	1.6 M	1.6 M	3.21 / 3.59	22959
110.5 K	4162	4172	0	50.0 M	48.2 M	1.8 M	1.8 M	3.62 / 3.77	21858
52.5 K	2011	2011	0	75.9 M	73.0 M	2.9 M	2.9 M	3.83 / 3.83	9652

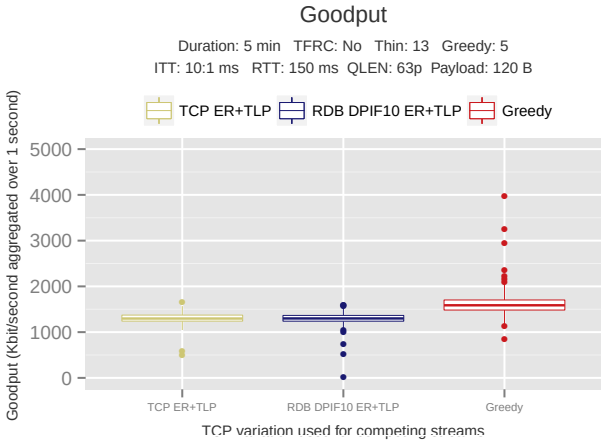
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
263	213	217	224	298	428	452	492	545	584	620	687	1266	2130	4386
267	213	217	224	312	431	459	521	569	602	631	777	1419	3764	3968
266	216	222	234	300	431	455	487	517	566	617	671	1082	1965	3603

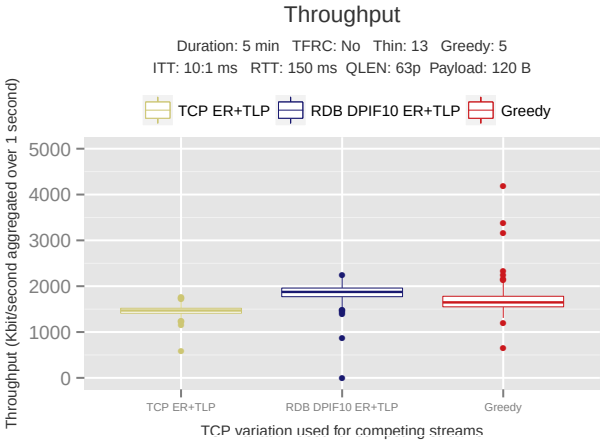
(c)

Figure A.3.51



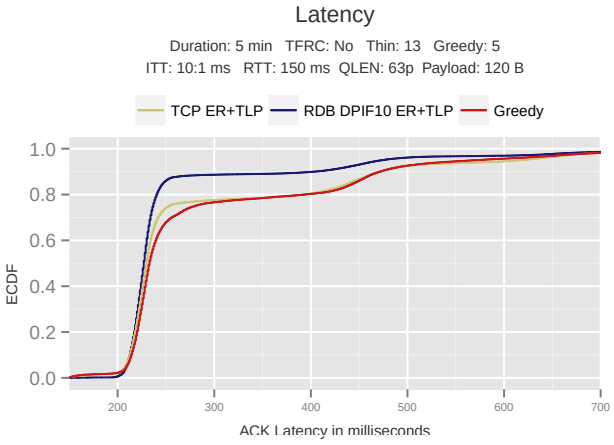
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	37 / 40 / 43	490/524/567	7/1/5/7/9/25
RDB DPIF10 ER+TLP	rdbsender	13	37 / 42 / 46	674/736/789	7/1/5/7/9/22
Greedy	zsender	5	32 / 34 / 35	1445/1447/1447	8/1/5/7/9/31

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
97.3 K	4256	4256	0	50.9 M	48.9 M	1.9 M	1.9 M	3.79 / 4.37	19839
90.5 K	4099	2154	54.1 K	66.5 M	48.4 M	18.1 M	1.8 M	4.48 / 4.53	9005
44.1 K	1993	1993	0	63.9 M	61.0 M	2.9 M	2.9 M	4.52 / 4.51	8767

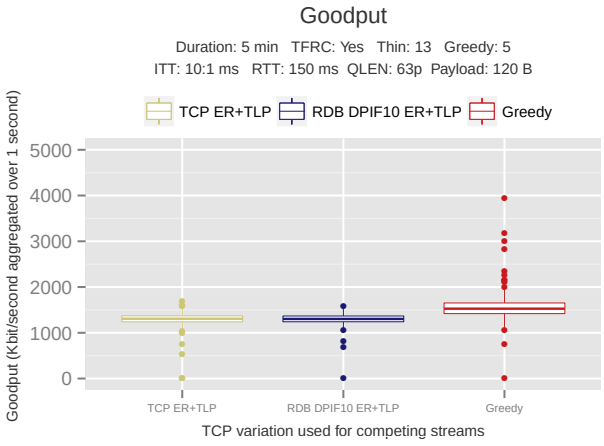
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
291	230	234	242	392	471	515	617	640	661	692	874	1340	1756	1996
262	227	230	234	241	404	449	474	495	607	661	803	1360	1816	2657
297	233	240	256	393	472	508	571	616	654	691	835	1826	6252	6739

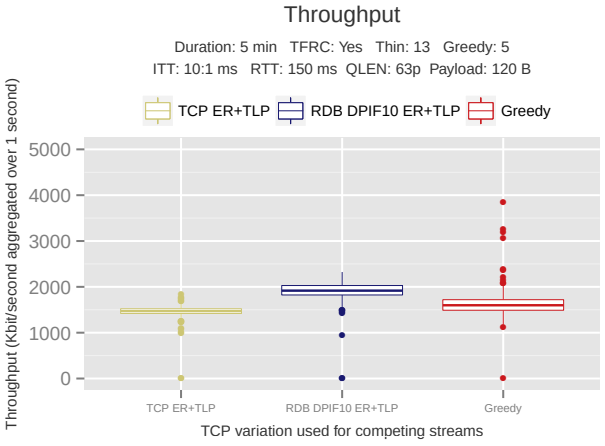
(c)

Figure A.3.52



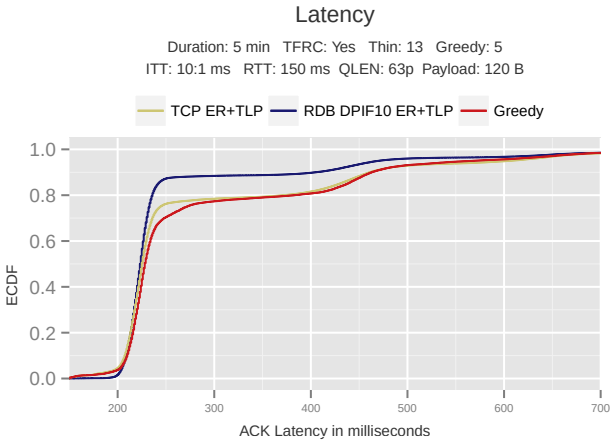
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	36 / 39 / 43	470/508/560	7/1/5/7/9/24
RDB DPIF10 ER+TLP	rdbsender	13	35 / 38 / 42	634/676/735	8/1/5/7/9/78
Greedy	zsender	5	33 / 35 / 38	1445/1446/1447	7/1/5/7/9/23

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
100.3 K	4282	4292	0	50.9 M	49.0 M	1.9 M	1.9 M	3.78 / 4.26	19968
101.0 K	4815	2383	66.3 K	68.1 M	48.2 M	19.9 M	1.8 M	4.67 / 4.77	10142
42.7 K	1926	1928	0	61.7 M	58.9 M	2.8 M	2.8 M	4.51 / 4.51	8344

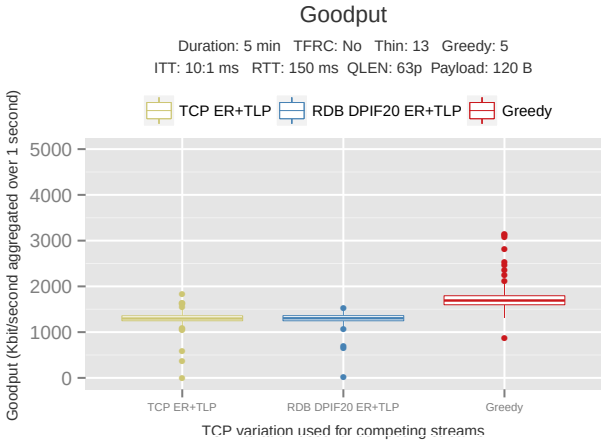
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
285	225	230	236	372	461	503	606	632	653	684	873	1368	3564	3850
261	223	227	230	236	405	445	470	501	618	663	873	1752	3835	3992
288	228	234	248	383	463	497	568	615	647	678	822	1474	2295	2649

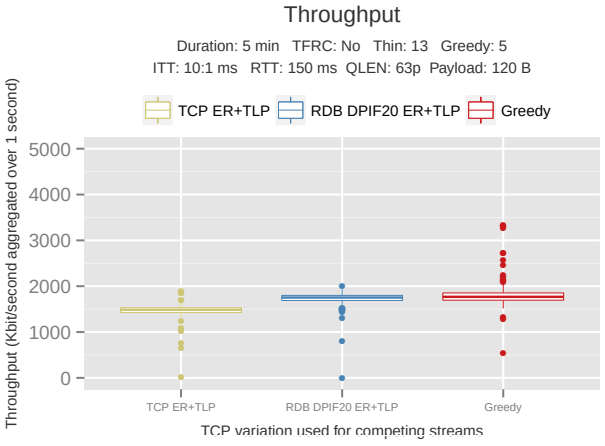
(c)

Figure A.3.53



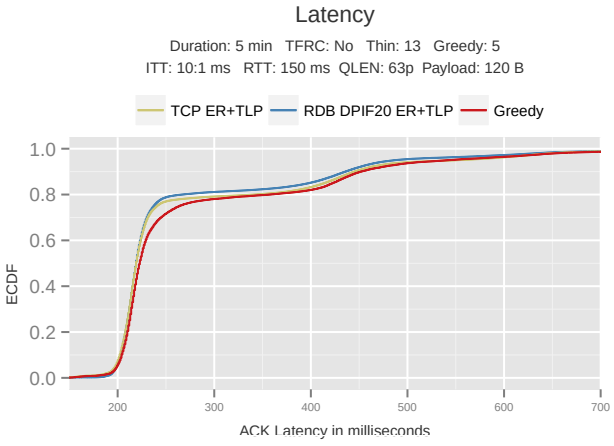
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	34 / 37 / 39	438/480/515	7/1/5/7/9/22
RDB DPIF20 ER+TLP	rdbsender	13	33 / 36 / 40	514/567/629	8/1/5/7/9/27
Greedy	zsender	5	31 / 32 / 33	1446/1447/1447	8/1/5/7/9/30

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
105.9 K	4151	4161	0	50.7 M	48.9 M	1.8 M	1.8 M	3.56 / 3.92	21093
107.3 K	4775	3795	29.1 K	60.6 M	48.4 M	12.2 M	1.9 M	4.37 / 4.45	19754
46.8 K	2044	2044	0	67.8 M	64.8 M	3.0 M	3.0 M	4.36 / 4.36	9117

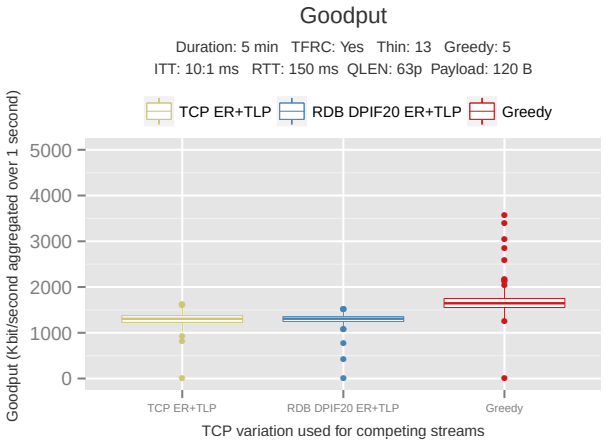
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
274	219	224	231	341	445	475	552	593	621	648	738	1270	3279	4029
267	219	223	230	267	436	459	487	527	588	633	716	1186	1682	2347
280	223	229	245	358	452	488	543	582	620	654	759	1299	1701	1895

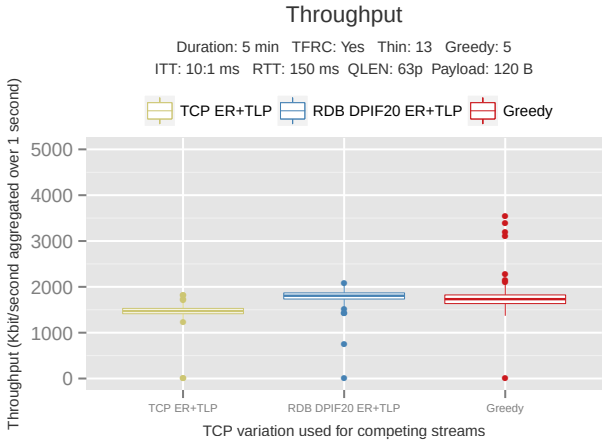
(c)

Figure A.3.54



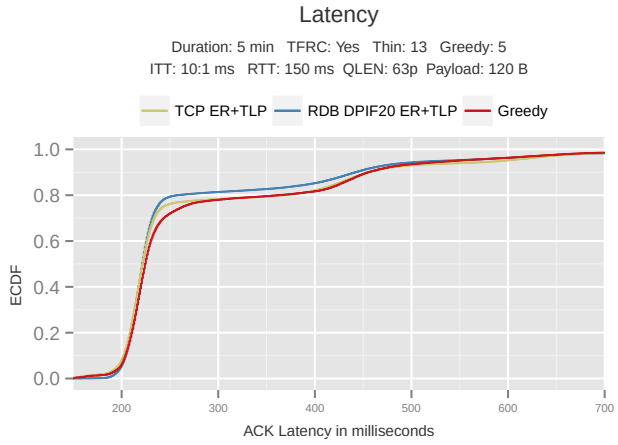
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	34 / 38 / 41	437/497/542	7/1/5/7/9/24
RDB DPIF20 ER+TLP	rdbsender	13	34 / 37 / 42	552/608/678	8/1/5/8/10/74
Greedy	zsender	5	30 / 33 / 34	1446/1447/1447	7/1/5/7/9/24

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
102.9 K	4471	4481	0	51.0 M	49.0 M	2.0 M	2.0 M	3.90 / 4.34	20487
104.3 K	5083	3553	40.1 K	63.3 M	48.3 M	15.0 M	2.0 M	4.82 / 4.87	17750
45.6 K	2029	2031	0	66.0 M	63.1 M	2.9 M	2.9 M	4.45 / 4.45	8679

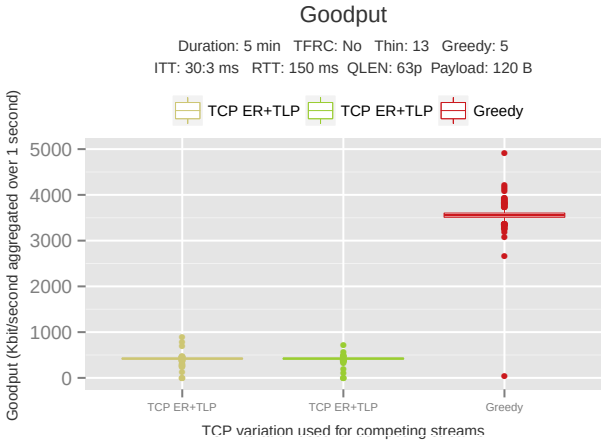
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
280	222	226	234	368	456	499	595	619	642	674	848	1354	1926	2574
273	221	226	232	258	442	472	531	588	628	671	870	1508	2281	2634
287	224	230	244	362	455	489	546	584	625	664	831	3355	7443	9065

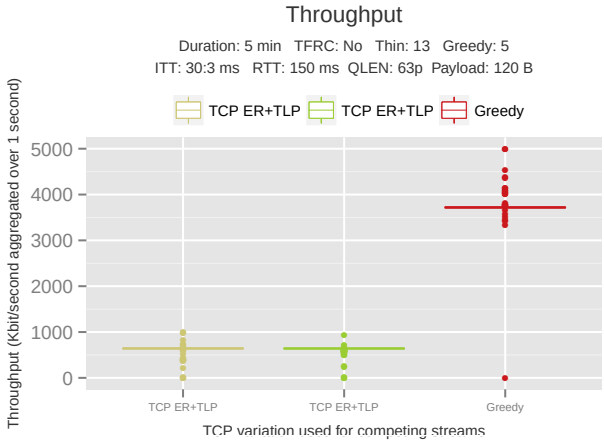
(c)

Figure A.3.55



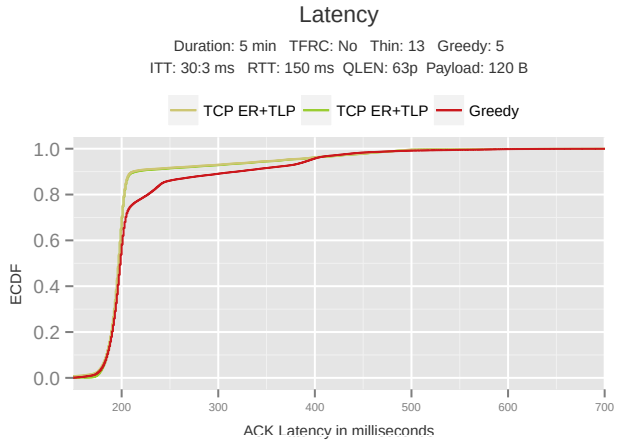
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	31 / 32 / 33	125/129/133	10/1/8/11/11/12
TCP ER+TLP	rdbsender	13	31 / 31 / 33	125/128/133	10/1/8/11/11/13
Greedy	zsender	5	15 / 16 / 16	1447/1447/1447	13/4/10/12/15/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
124.3 K	1479	1483	0	16.0 M	15.8 M	193.5 K	193.5 K	1.21 / 1.19	10443
122.2 K	1460	1470	0	15.7 M	15.5 M	181.9 K	181.9 K	1.16 / 1.19	10678
94.4 K	1438	1438	0	136.6 M	134.5 M	2.1 M	2.1 M	1.52 / 1.52	12361

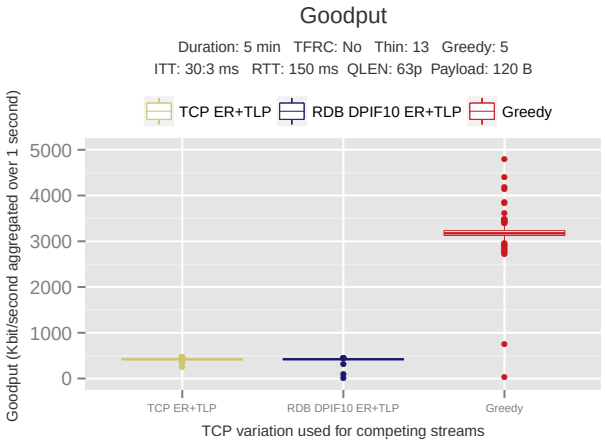
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
213	197	199	200	203	212	299	364	395	424	453	482	608	7416	9889
213	197	199	200	203	215	306	368	397	426	455	483	635	1418	17310
223	199	201	204	227	319	378	395	402	418	441	488	637	787	999

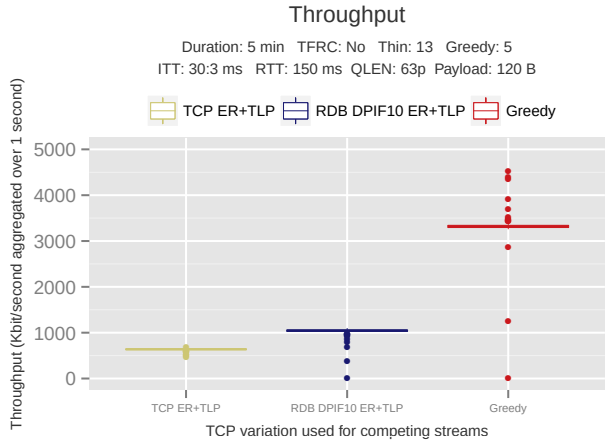
(c)

Figure A.3.56



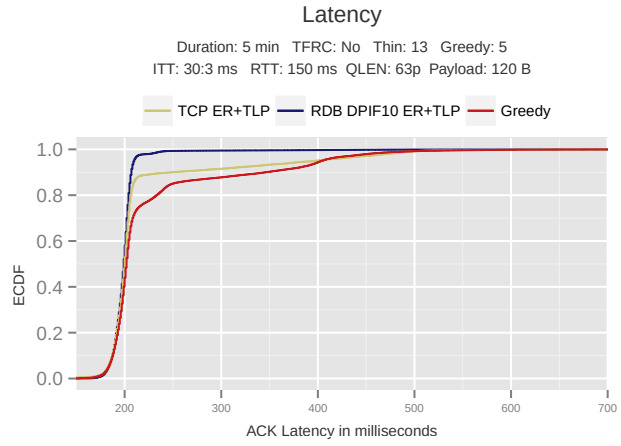
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	31 / 31 / 32	127/129/131	9/1/7/10/11/12
RDB DPIF10 ER+TLP	rdbsender	13	32 / 32 / 33	256/262/269	9/1/7/10/11/13
Greedy	zsender	5	17 / 18 / 18	1447/1447/1447	12/3/9/11/13/30

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
124.5 K	1781	1781	0	16.1 M	15.9 M	225.0 K	225.0 K	1.40 / 1.43	12334
119.2 K	1710	114	119.0 K	31.2 M	15.6 M	15.6 M	15.2 K	1.43 / 1.43	592
84.6 K	1498	1500	0	122.5 M	120.3 M	2.2 M	2.2 M	1.77 / 1.77	11885

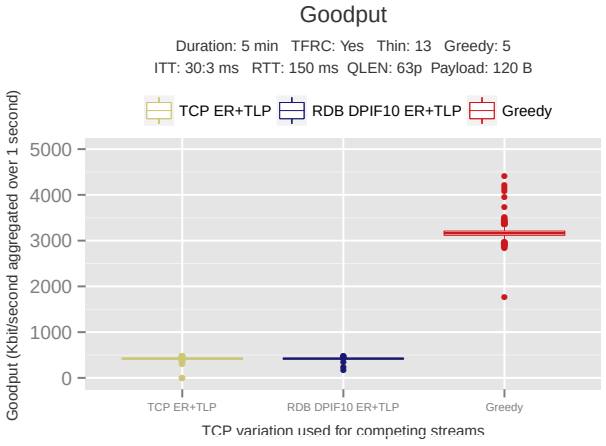
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
218	200	202	204	206	250	342	398	422	447	471	496	695	1143	1401
200	199	201	202	204	207	208	210	211	213	225	238	499	1151	1576
228	202	204	208	233	346	391	404	413	429	452	488	647	878	1150

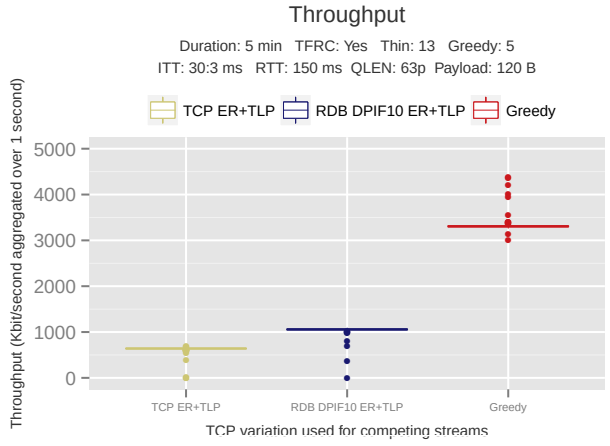
(c)

Figure A.3.57



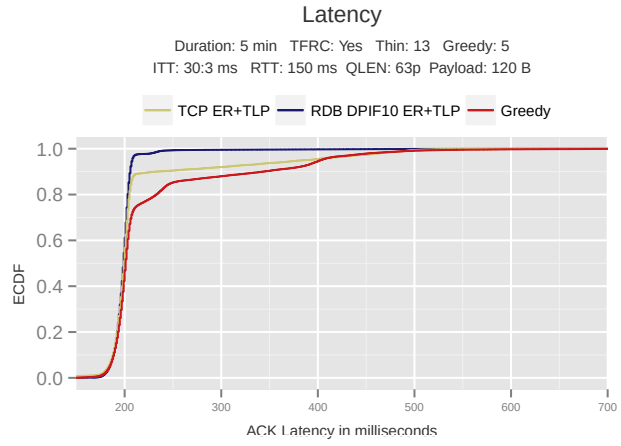
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	31 / 31 / 32	127/128/131	9/1/8/11/11/12
RDB DPIF10 ER+TLP	rdbsender	13	30 / 30 / 30	242/244/247	10/1/10/11/12/31
Greedy	zsender	5	17 / 18 / 18	1446/1447/1448	11/3/9/11/13/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
124.6 K	1642	1652	0	16.0 M	15.8 M	205.3 K	205.3 K	1.28 / 1.32	11839
128.2 K	1854	93	128.1 K	31.3 M	15.7 M	15.6 M	11.8 K	1.44 / 1.45	554
84.0 K	1504	1504	0	121.6 M	119.4 M	2.2 M	2.2 M	1.79 / 1.79	11706

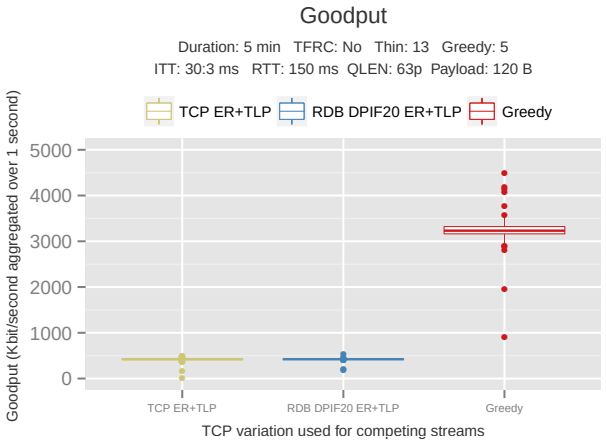
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
216	199	201	203	205	233	331	388	409	436	463	490	587	824	3235
200	199	200	202	203	206	207	208	209	210	228	237	502	715	1044
228	201	203	207	233	343	390	403	411	430	454	492	702	928	999

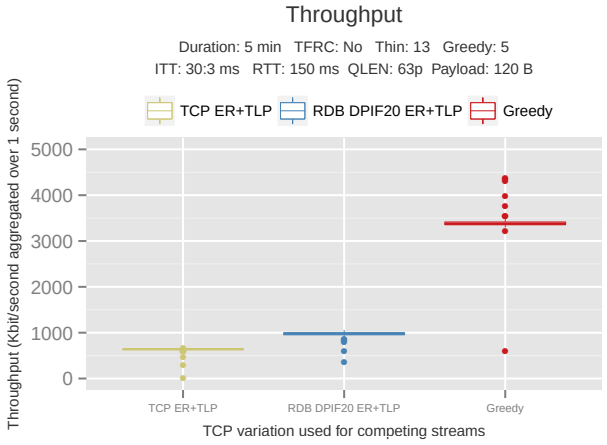
(c)

Figure A.3.58



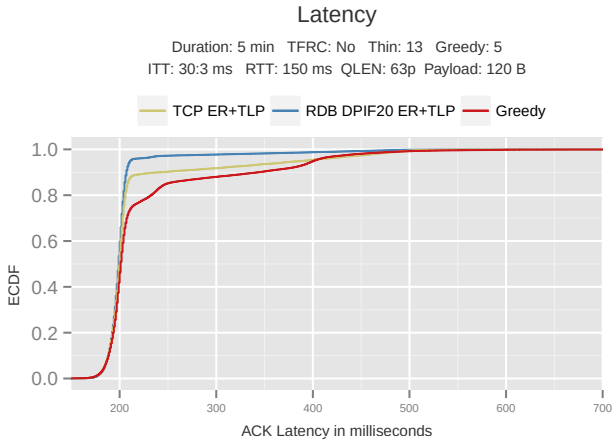
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	31 / 31 / 32	127/129/131	9/1/7/11/11/13
RDB DPIF20 ER+TLP	rdbsender	13	31 / 32 / 32	228/237/244	9/1/7/10/11/14
Greedy	zsender	5	16 / 17 / 19	1447/1447/1447	12/1/9/11/14/39

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
124.6 K	1691	1701	0	16.0 M	15.8 M	208.2 K	208.2 K	1.30 / 1.36	12132
121.3 K	1730	478	99.7 K	28.7 M	15.7 M	13.1 M	58.4 K	1.38 / 1.43	3245
86.1 K	1549	1549	0	124.6 M	122.4 M	2.2 M	2.2 M	1.80 / 1.80	11965

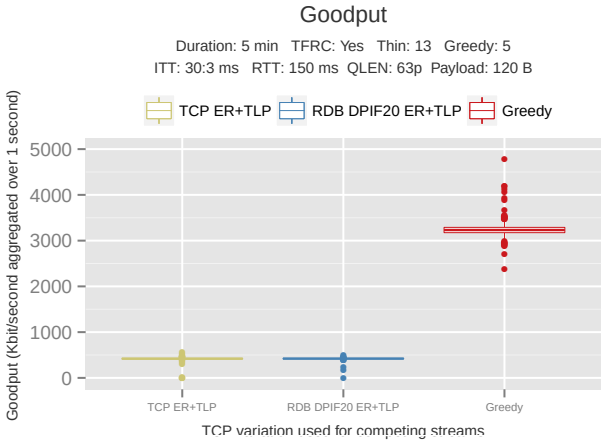
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
217	200	201	203	206	241	336	390	414	438	464	489	564	802	1227
204	199	201	202	204	207	209	211	219	239	327	425	543	709	999
229	201	204	208	233	343	389	401	409	423	447	488	658	5485	5698

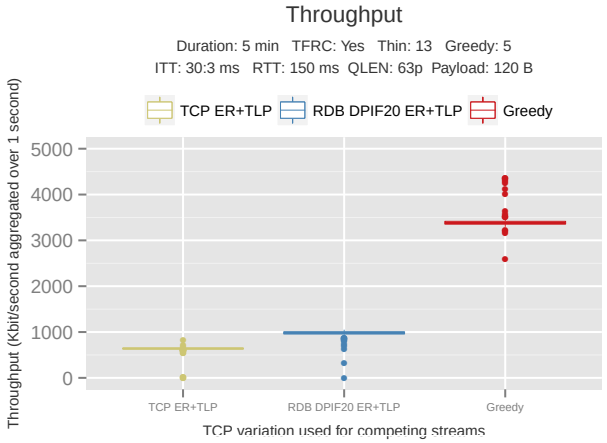
(c)

Figure A.3.59



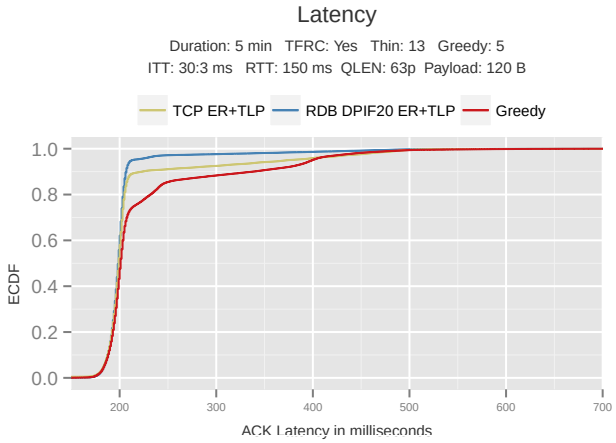
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	31 / 31 / 32	126/128/131	10/1/8/11/11/12
RDB DPIF20 ER+TLP	rdbsender	13	31 / 31 / 31	226/229/234	11/1/10/11/12/26
Greedy	zsender	5	17 / 17 / 18	1447/1447/1448	12/4/9/11/13/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
124.5 K	1534	1542	0	16.0 M	15.8 M	189.1 K	189.1 K	1.19 / 1.23	11232
124.4 K	1617	489	101.0 K	28.5 M	15.7 M	12.9 M	58.9 K	1.26 / 1.30	3410
86.2 K	1481	1481	0	124.8 M	122.7 M	2.1 M	2.1 M	1.72 / 1.72	11804

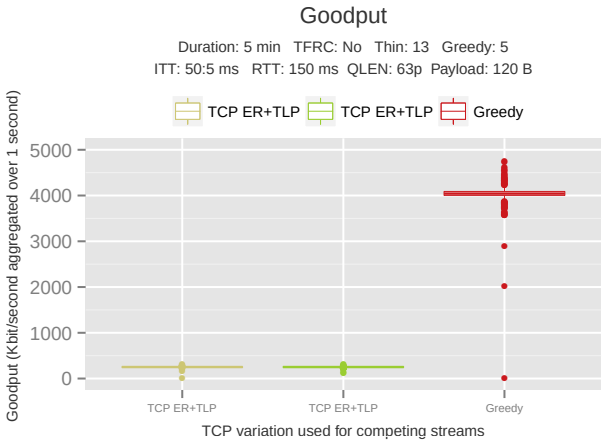
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
215	199	201	203	205	223	314	375	402	430	459	486	598	827	7610
204	199	200	202	204	207	209	213	228	243	339	434	593	981	1402
227	201	203	208	232	338	386	400	407	422	445	481	626	739	999

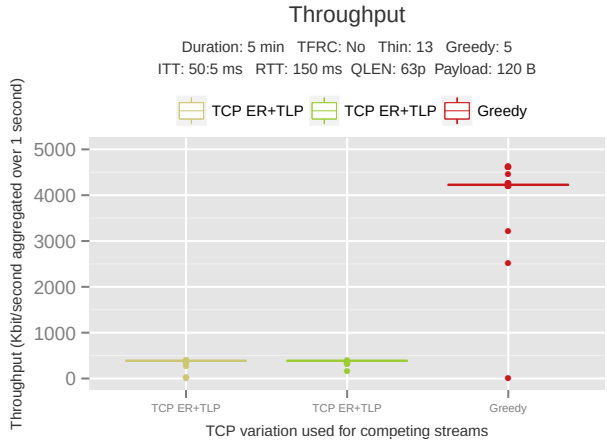
(c)

Figure A.3.60



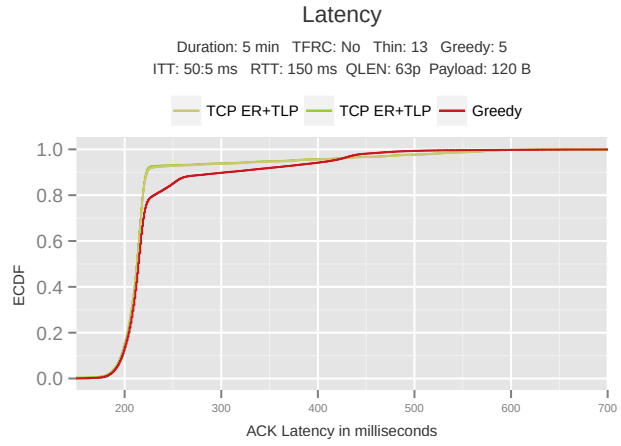
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	50 / 50 / 50	121/122/123	8/1/7/9/9/10
TCP ER+TLP	rdbsender	13	50 / 50 / 50	121/122/123	8/1/7/9/9/10
Greedy	zsender	5	13 / 14 / 15	1446/1447/1447	15/3/12/15/18/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
78.3 K	941	951	0	9.6 M	9.5 M	113.6 K	113.6 K	1.19 / 1.20	5245
77.4 K	906	907	0	9.5 M	9.3 M	109.9 K	109.9 K	1.16 / 1.17	4997
106.6 K	1286	1287	0	154.3 M	152.4 M	1.9 M	1.9 M	1.20 / 1.21	12470

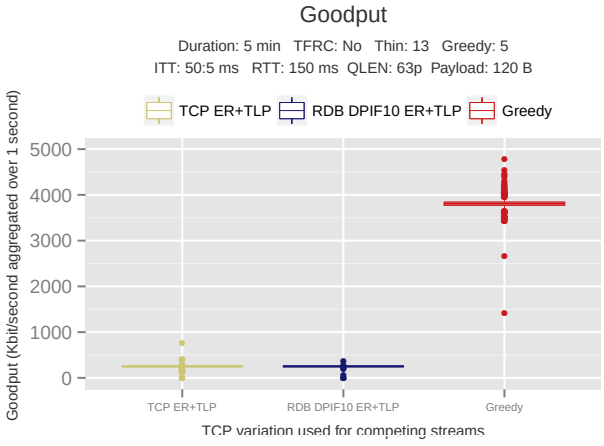
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
227	213	215	217	219	223	263	372	422	471	519	564	680	1152	1181
226	213	215	217	219	223	244	367	419	467	516	564	648	1030	1203
235	214	217	220	230	305	375	414	425	433	446	479	657	898	1153

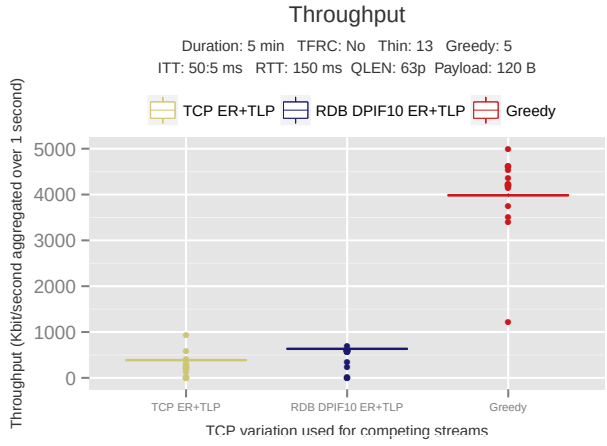
(c)

Figure A.3.61



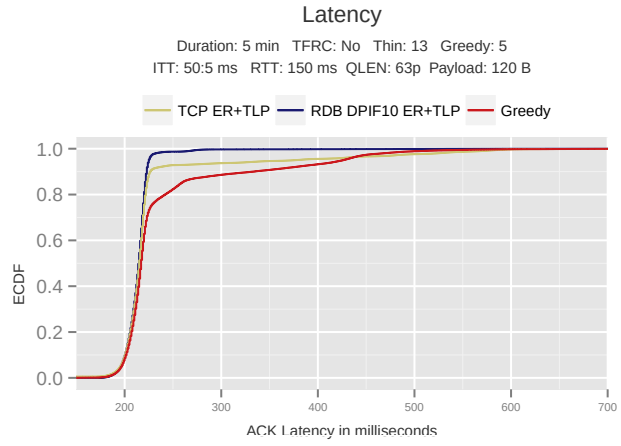
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	50 / 51 / 52	122/124/127	8/1/7/9/9/10
RDB DPIF10 ER+TLP	rdbsender	13	51 / 51 / 52	243/244/246	8/1/7/9/9/10
Greedy	zsender	5	14 / 15 / 15	1447/1447/1447	15/5/12/14/17/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
76.8 K	908	918	0	9.5 M	9.4 M	113.8 K	113.8 K	1.19 / 1.18	5073
75.7 K	833	57	75.6 K	18.5 M	9.2 M	9.2 M	5.9 K	1.09 / 1.10	175
100.8 K	1274	1274	0	145.8 M	144.0 M	1.8 M	1.8 M	1.26 / 1.26	12852

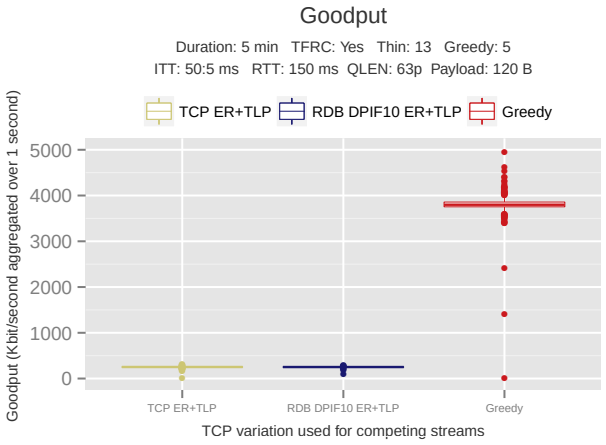
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
230	215	217	219	222	227	265	377	427	476	524	570	687	3267	7751
215	215	217	218	220	223	224	225	226	228	232	267	531	2216	7758
241	217	220	223	242	333	396	426	435	445	468	508	681	830	1084

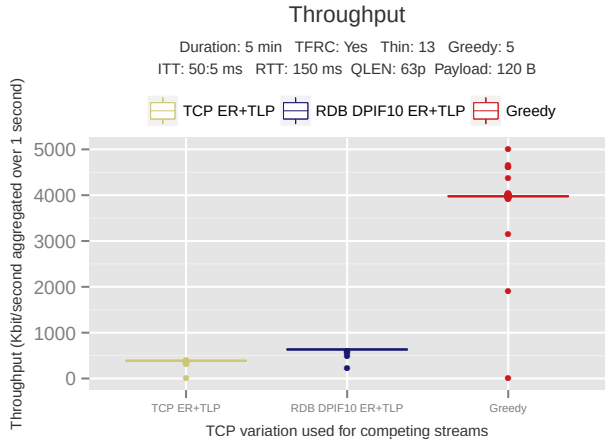
(c)

Figure A.3.62



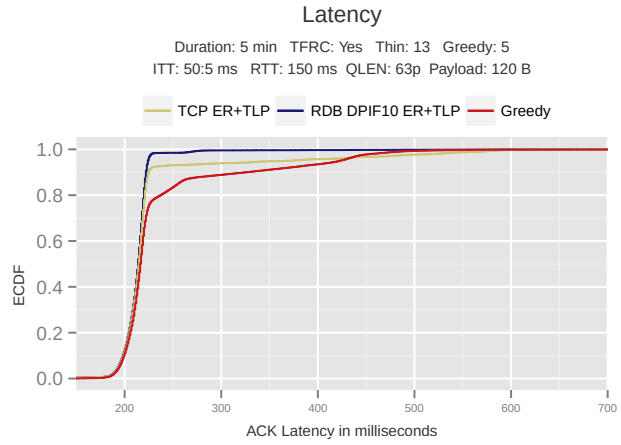
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	50 / 50 / 50	121/122/123	8/1/7/9/9/10
RDB DPIF10 ER+TLP	rdbsender	13	50 / 50 / 50	240/241/241	10/1/9/9/10/19
Greedy	zsender	5	14 / 15 / 15	1446/1447/1447	15/1/12/14/17/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
78.3 K	912	912	0	9.6 M	9.5 M	109.7 K	109.7 K	1.14 / 1.16	5028
77.7 K	916	61	77.5 K	18.7 M	9.3 M	9.3 M	7.3 K	1.18 / 1.18	301
100.6 K	1313	1315	0	145.6 M	143.7 M	1.9 M	1.9 M	1.31 / 1.30	12692

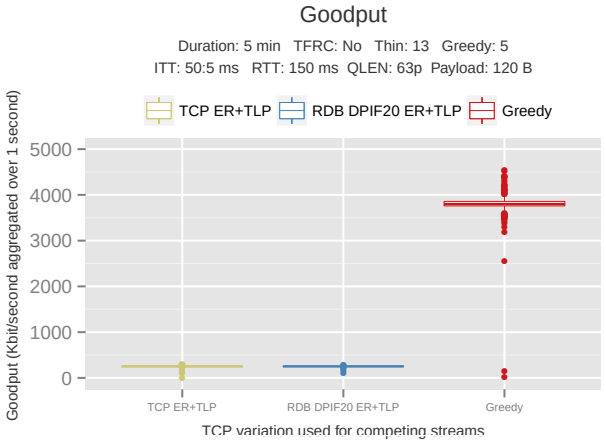
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
228	215	217	219	221	225	247	368	422	472	521	567	648	906	1352
214	214	216	218	220	222	223	224	225	226	228	270	566	735	904
239	216	219	222	237	326	389	423	431	440	456	484	660	1655	1672

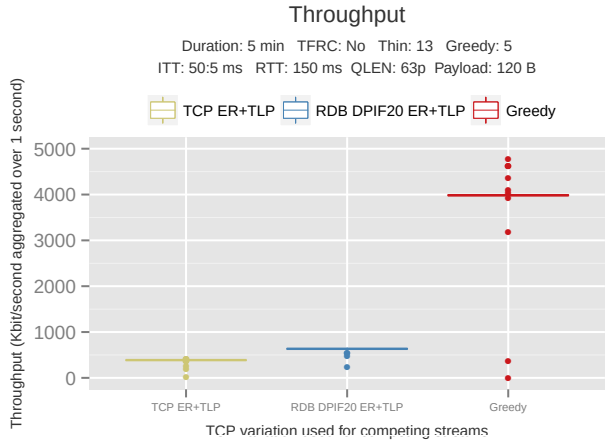
(c)

Figure A.3.63



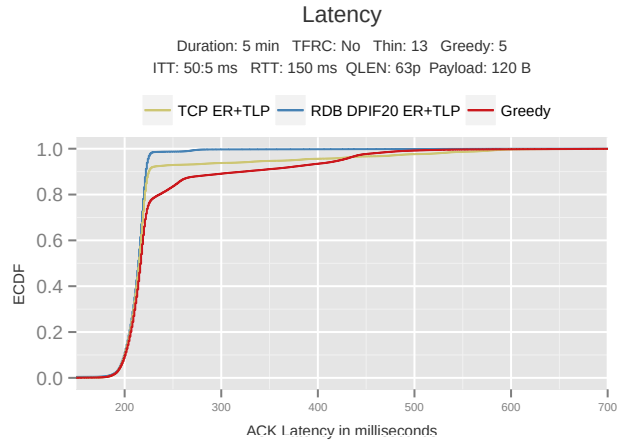
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	50 / 50 / 50	122/123/123	8/1/7/9/9/10
RDB DPIF20 ER+TLP	rdbsender	13	50 / 50 / 51	242/243/245	8/1/8/9/9/10
Greedy	zsender	5	14 / 15 / 16	1447/1447/1448	15/4/12/14/17/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
78.0 K	929	939	0	9.6 M	9.4 M	112.3 K	112.3 K	1.17 / 1.19	5114
76.8 K	824	56	76.6 K	18.7 M	9.3 M	9.3 M	7.0 K	1.07 / 1.07	187
100.5 K	1294	1296	0	145.4 M	143.6 M	1.9 M	1.9 M	1.29 / 1.29	12283

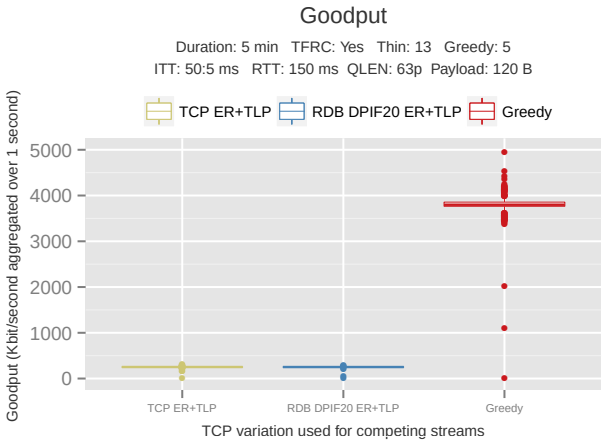
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
229	215	217	219	221	225	258	374	426	475	524	571	695	1153	1579
214	214	216	218	220	222	223	224	225	226	227	266	528	697	1126
239	217	219	222	236	323	391	423	432	440	458	487	679	924	1150

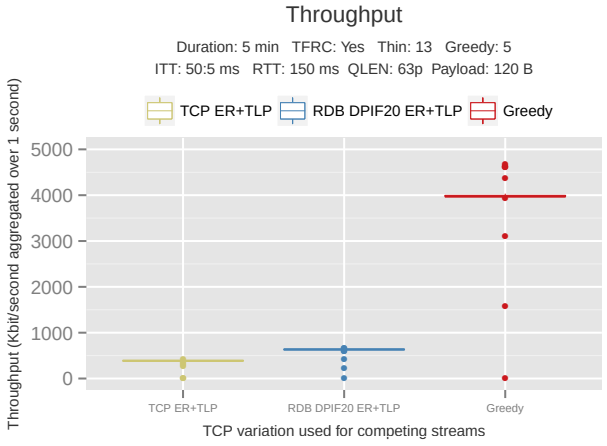
(c)

Figure A.3.64



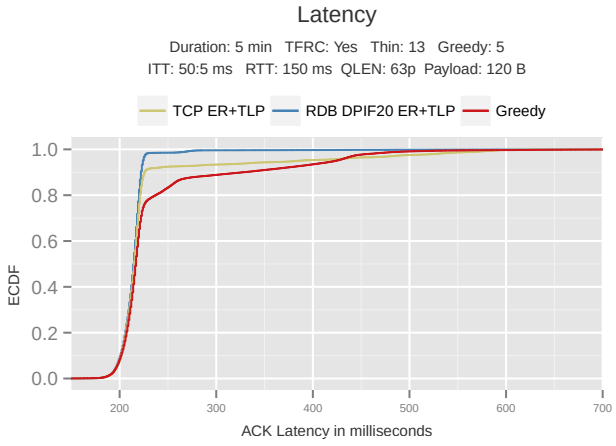
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	50 / 50 / 50	122/123/123	8/1/7/9/9/10
RDB DPIF20 ER+TLP	rdbsender	13	50 / 50 / 50	240/241/241	10/1/9/9/10/82
Greedy	zsender	5	14 / 15 / 15	1446/1447/1447	15/3/12/14/17/36

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
78.1 K	970	981	0	9.6 M	9.5 M	117.4 K	117.4 K	1.22 / 1.24	5475
77.4 K	897	67	77.3 K	18.6 M	9.3 M	9.3 M	7.2 K	1.16 / 1.16	259
100.6 K	1299	1301	0	145.5 M	143.6 M	1.9 M	1.9 M	1.29 / 1.29	12554

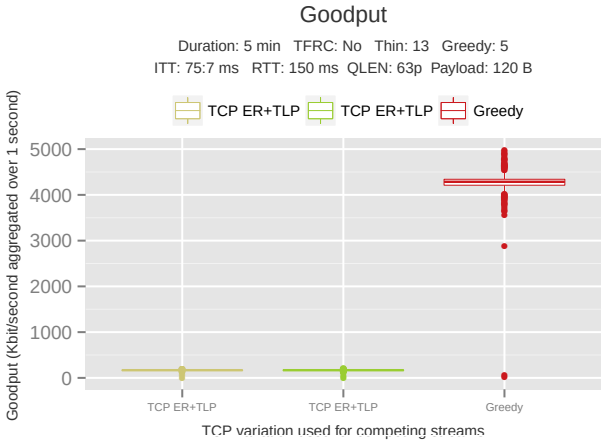
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
231	215	217	219	221	226	283	384	433	481	527	570	645	1150	1152
215	215	216	218	220	222	223	224	225	226	227	268	566	1151	1181
240	217	219	222	237	327	392	424	432	441	457	492	680	848	1150

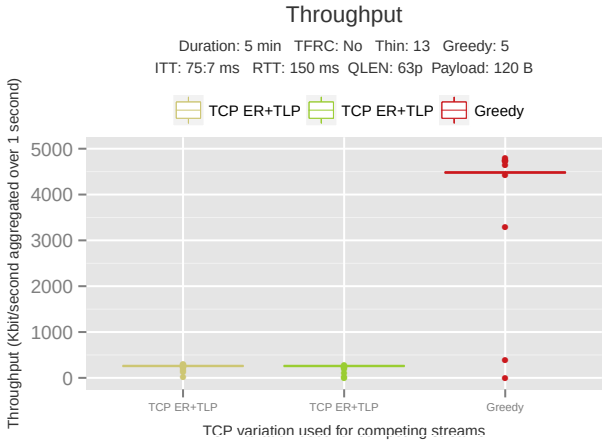
(c)

Figure A.3.65



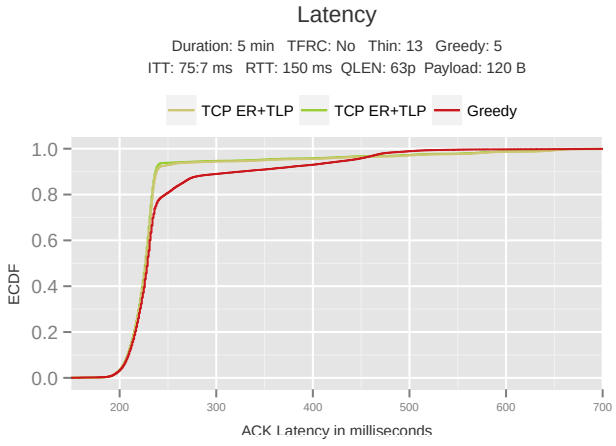
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	74 / 74 / 74	120/120/121	7/1/7/8/8/10
TCP ER+TLP	rdbsender	13	74 / 74 / 74	120/120/120	7/1/7/8/8/10
Greedy	zsender	5	12 / 13 / 14	1447/1447/1447	18/1/14/17/21/47

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.8 K	658	672	0	6.4 M	6.3 M	79.4 K	79.4 K	1.24 / 1.24	2807
52.1 K	599	610	0	6.3 M	6.2 M	71.6 K	71.6 K	1.14 / 1.15	2654
112.6 K	1168	1170	0	163.0 M	161.3 M	1.7 M	1.7 M	1.03 / 1.04	13710

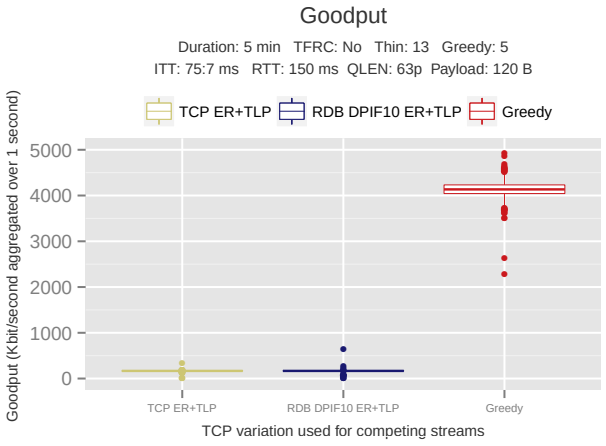
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
241	227	230	232	234	238	252	351	425	497	570	641	767	1151	1151
239	227	229	231	234	237	240	336	412	487	560	634	672	1152	1153
250	229	231	235	247	325	400	438	452	462	474	504	693	1460	1669

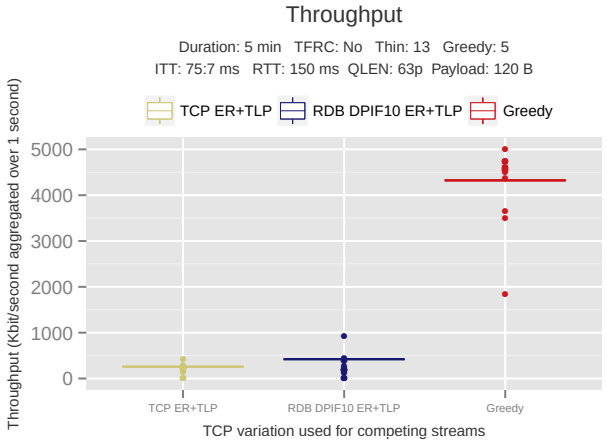
(c)

Figure A.3.66



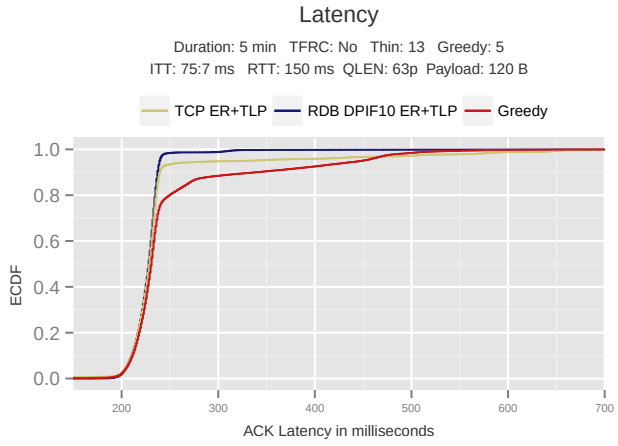
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	74 / 75 / 77	120/121/123	7/1/7/8/8/10
RDB DPIF10 ER+TLP	rdbsender	13	75 / 77 / 78	240/242/245	7/1/7/8/8/10
Greedy	zsender	5	13 / 14 / 14	1446/1447/1447	17/7/13/16/20/44

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.4 K	606	610	0	6.3 M	6.3 M	71.2 K	71.2 K	1.12 / 1.16	2587
50.4 K	568	51	50.2 K	12.2 M	6.1 M	6.1 M	6.4 K	1.11 / 1.13	90
109.1 K	1162	1162	0	157.8 M	156.1 M	1.7 M	1.7 M	1.07 / 1.07	13472

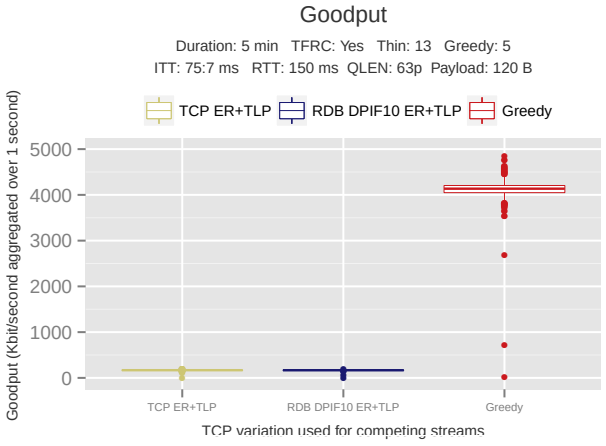
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
242	229	231	233	236	240	245	329	411	488	564	638	708	3272	7758
229	228	230	233	235	237	238	239	240	241	244	304	638	7726	7746
253	231	234	237	250	339	409	448	460	470	486	520	672	764	999

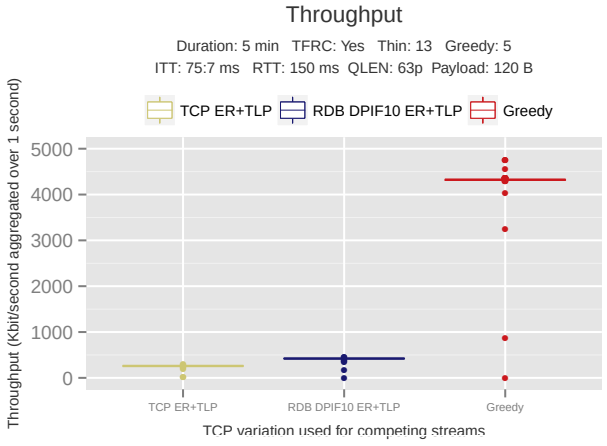
(c)

Figure A.3.67



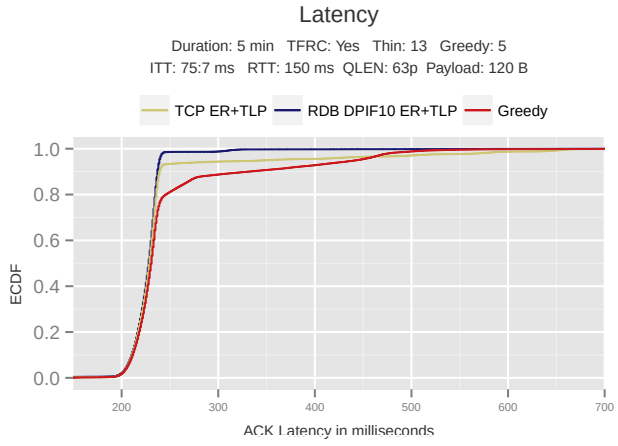
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	74 / 74 / 74	120/120/120	7/1/7/8/8/10
RDB DPIF10 ER+TLP	rdbsender	13	74 / 75 / 75	240/240/240	9/1/8/8/10/23
Greedy	zsender	5	13 / 14 / 14	1446/1447/1447	17/6/13/16/20/40

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
53.0 K	629	644	0	6.4 M	6.3 M	76.1 K	76.1 K	1.18 / 1.19	2843
51.8 K	614	40	51.7 K	12.4 M	6.2 M	6.2 M	4.6 K	1.18 / 1.19	127
108.7 K	1186	1188	0	157.3 M	155.6 M	1.7 M	1.7 M	1.09 / 1.09	13242

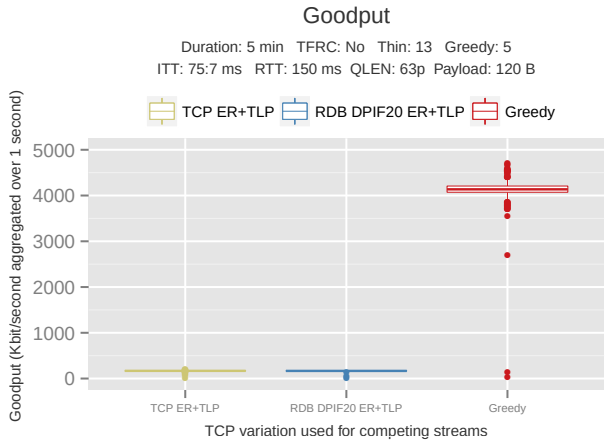
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
242	229	231	234	236	239	243	351	426	498	571	641	713	1152	1153
228	228	231	233	235	237	238	239	240	241	242	306	586	796	926
252	231	233	236	246	331	404	444	456	466	477	507	693	868	1153

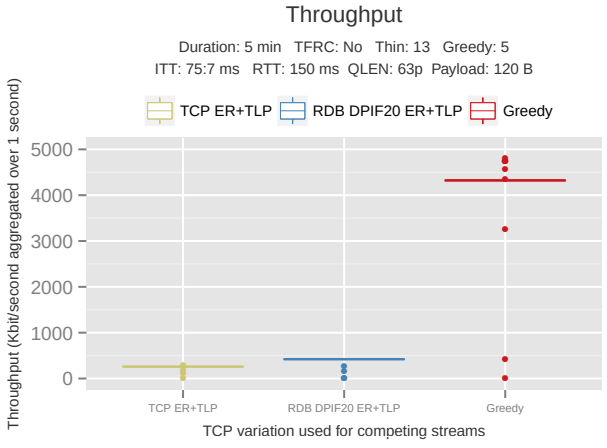
(c)

Figure A.3.68



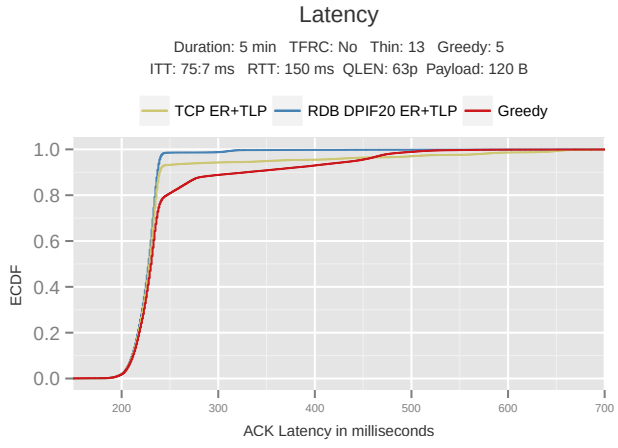
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	74 / 74 / 74	120/120/120	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	13	75 / 75 / 76	240/241/242	7/1/7/8/8/10
Greedy	zsender	5	13 / 14 / 14	1446/1447/1448	17/5/13/16/20/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.9 K	655	667	0	6.4 M	6.3 M	78.8 K	78.8 K	1.24 / 1.24	2874
51.4 K	595	50	51.4 K	12.4 M	6.2 M	6.2 M	4.4 K	1.15 / 1.16	112
108.7 K	1179	1181	0	157.3 M	155.6 M	1.7 M	1.7 M	1.09 / 1.08	13149

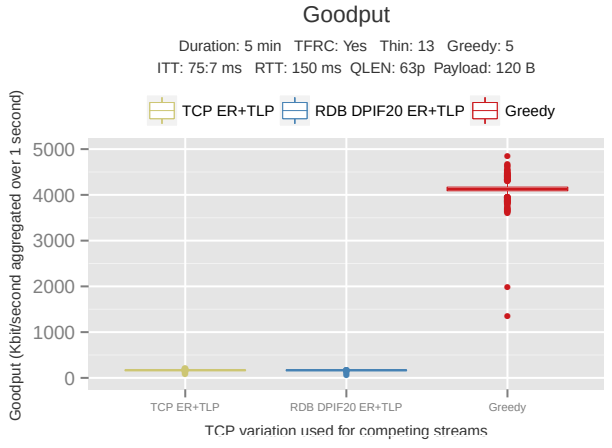
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
243	229	231	233	236	239	244	355	429	500	572	642	728	1152	1152
228	228	230	233	235	237	238	239	240	241	242	306	602	1152	1153
251	231	233	236	247	329	401	442	456	465	476	503	651	926	1150

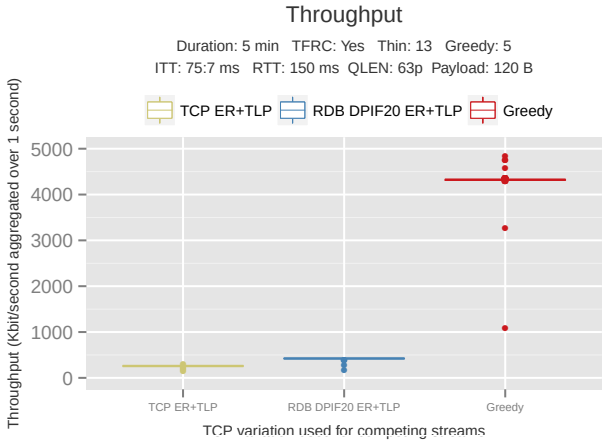
(c)

Figure A.3.69



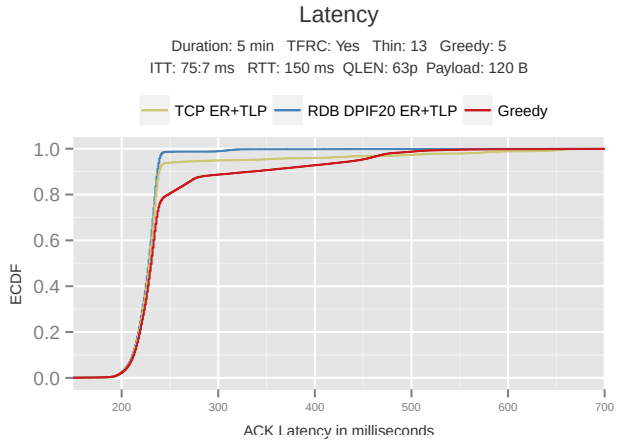
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	74 / 74 / 74	120/120/121	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	13	74 / 75 / 75	240/240/240	10/1/8/8/11/22
Greedy	zsender	5	13 / 14 / 14	1446/1447/1447	17/5/13/16/19/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.8 K	579	581	0	6.3 M	6.3 M	69.7 K	69.7 K	1.09 / 1.10	2596
51.7 K	581	30	51.7 K	12.4 M	6.2 M	6.2 M	3.6 K	1.12 / 1.12	100
108.7 K	1165	1165	0	157.3 M	155.6 M	1.7 M	1.7 M	1.07 / 1.07	13386

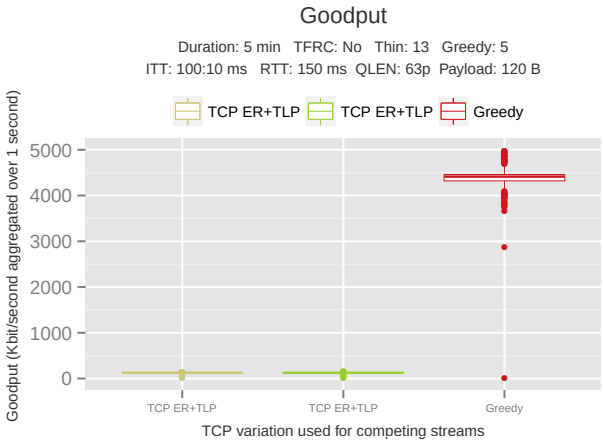
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
241	229	231	233	236	239	242	315	404	482	558	633	679	891	935
228	228	230	233	235	237	238	239	240	240	242	303	567	699	848
252	231	233	236	248	334	405	446	457	466	478	509	692	754	999

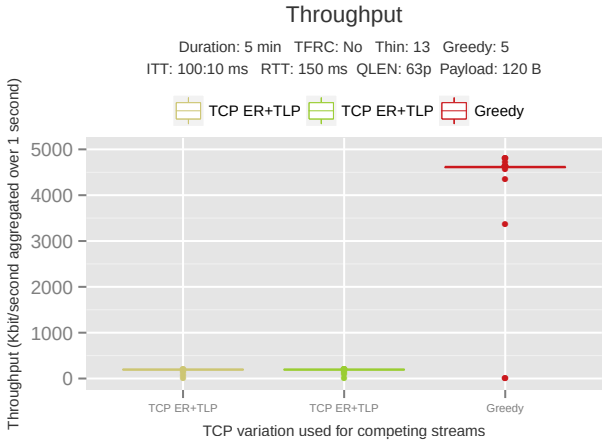
(c)

Figure A.3.70



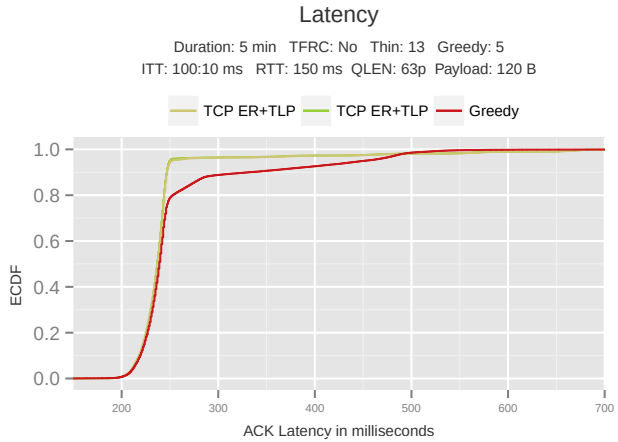
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	99 / 99 / 99	120/120/120	7/1/7/7/7/10
TCP ER+TLP	rdbsender	13	98 / 99 / 100	120/120/120	7/1/7/7/7/10
Greedy	zsender	5	12 / 13 / 13	1446/1447/1447	19/6/15/18/22/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
39.5 K	353	366	0	4.7 M	4.7 M	42.7 K	42.7 K	0.89 / 0.89	1332
39.0 K	342	355	0	4.7 M	4.6 M	41.4 K	41.4 K	0.88 / 0.88	1289
115.7 K	1081	1083	0	167.4 M	165.9 M	1.6 M	1.6 M	0.93 / 0.93	13744

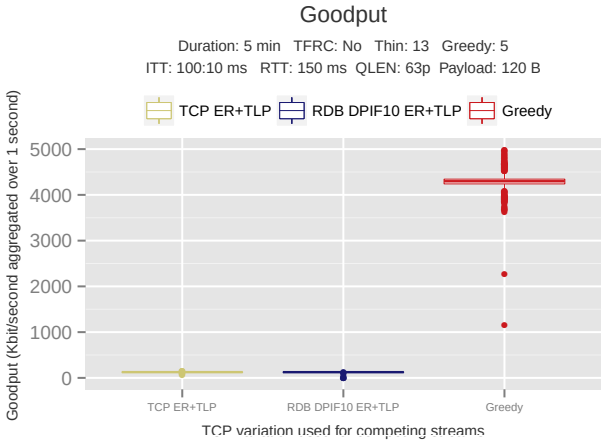
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
245	237	240	242	244	247	249	252	268	374	486	617	772	1153	1453
245	237	240	242	244	247	248	250	257	368	478	599	762	1152	1417
259	239	242	245	253	332	408	451	469	480	490	516	671	778	1151

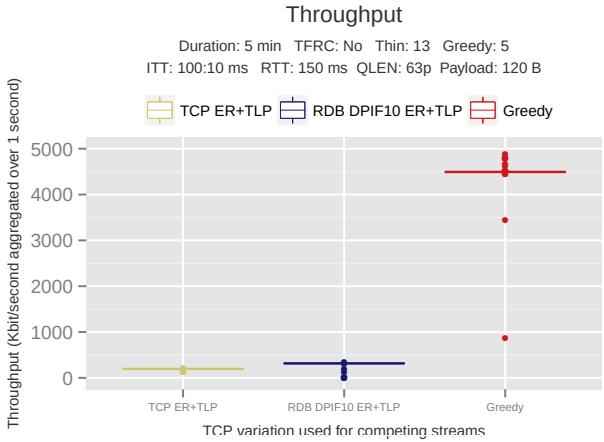
(c)

Figure A.3.71



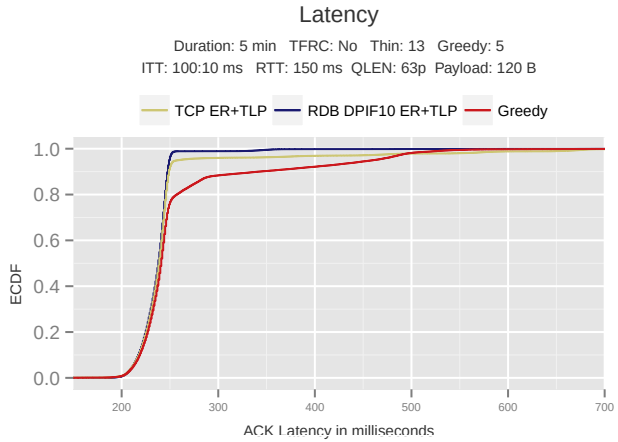
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	98 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF10 ER+TLP	rdbsender	13	100 / 101 / 101	240/240/240	7/1/7/7/7/10
Greedy	zsender	5	11 / 13 / 14	1446/1447/1447	18/6/15/18/21/4

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
39.6 K	408	423	0	4.8 M	4.7 M	50.8 K	50.8 K	1.03 / 1.03	1473
38.4 K	373	42	38.3 K	9.2 M	4.6 M	4.6 M	2.2 K	0.93 / 0.97	50
112.9 K	1125	1125	0	163.4 M	161.8 M	1.6 M	1.6 M	1.00 / 1.00	13797

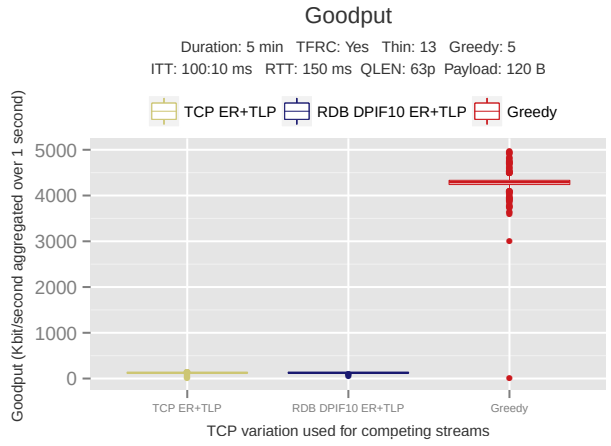
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
248	239	242	244	246	249	251	259	301	423	544	654	780	1068	1347
238	238	241	243	245	248	249	249	250	251	252	327	594	3255	3255
262	241	244	247	257	342	421	461	475	485	497	527	700	811	999

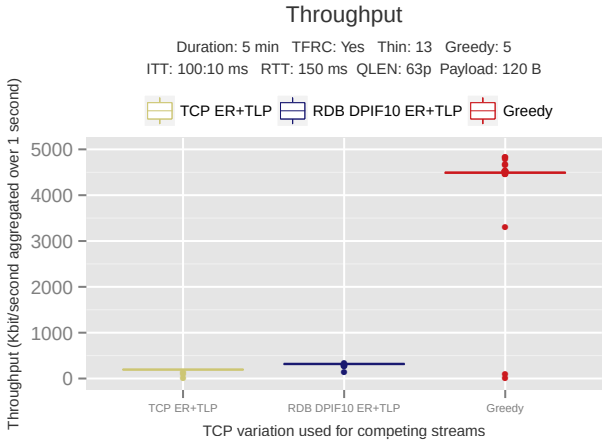
(c)

Figure A.3.72



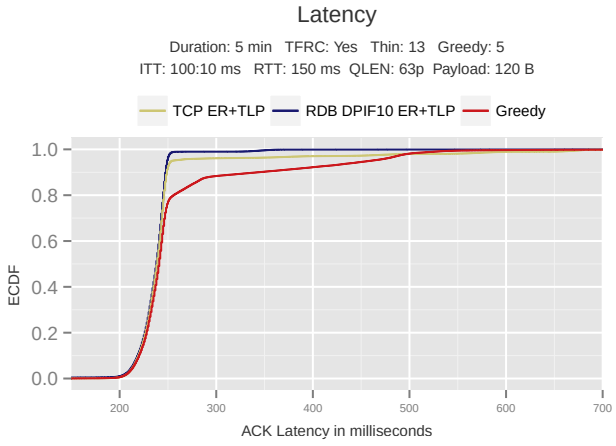
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	98 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF10 ER+TLP	rdbsender	13	99 / 99 / 100	240/240/240	10/1/8/9/11/27
Greedy	zsender	5	13 / 13 / 14	1446/1446/1447	18/7/15/18/21/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
39.5 K	385	406	0	4.7 M	4.7 M	47.5 K	47.5 K	0.97 / 0.97	1429
38.8 K	365	20	38.8 K	9.3 M	4.7 M	4.7 M	2.4 K	0.94 / 0.94	46
112.8 K	1119	1121	0	163.2 M	161.6 M	1.6 M	1.6 M	0.99 / 0.99	13844

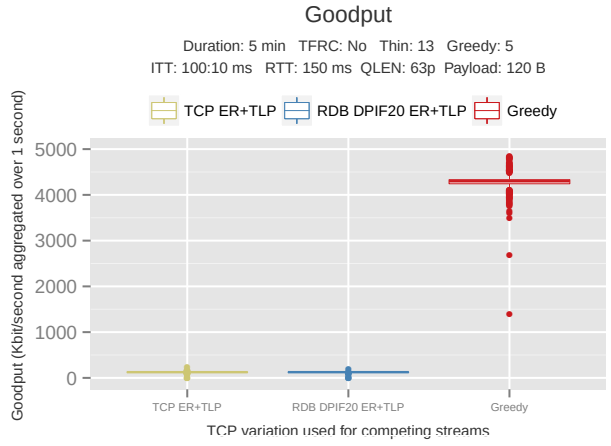
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
247	239	242	244	246	249	251	254	288	395	527	646	758	1151	1153
237	238	241	243	245	248	249	249	250	251	252	320	469	776	861
262	241	244	247	256	343	420	459	475	486	497	526	720	940	1153

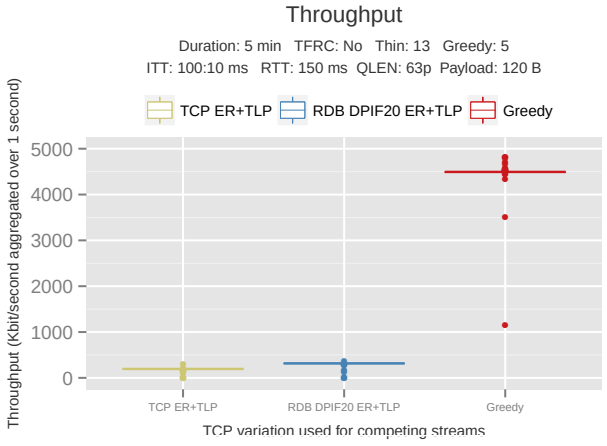
(c)

Figure A.3.73



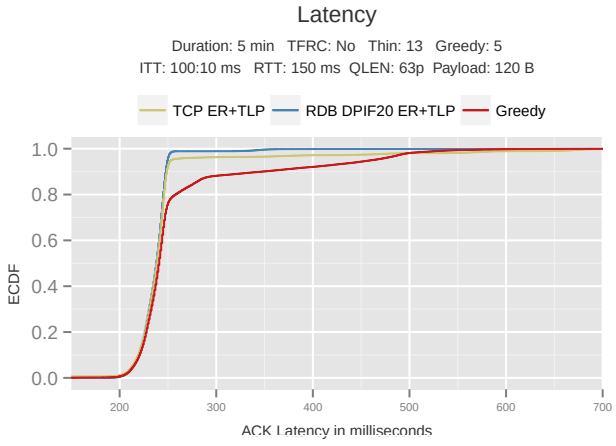
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	99 / 100 / 102	120/121/124	7/1/7/7/7/10
RDB DPIF20 ER+TLP	rdbsender	13	100 / 101 / 103	240/240/243	7/1/7/7/7/10
Greedy	zsender	5	12 / 13 / 14	1447/1447/1447	18/6/15/18/21/4

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
39.3 K	375	392	0	4.7 M	4.7 M	46.1 K	46.1 K	0.95 / 0.95	1342
38.3 K	384	38	38.2 K	9.2 M	4.6 M	4.6 M	2.4 K	0.96 / 1.00	49
113.0 K	1085	1085	0	163.5 M	161.9 M	1.6 M	1.6 M	0.96 / 0.96	13956

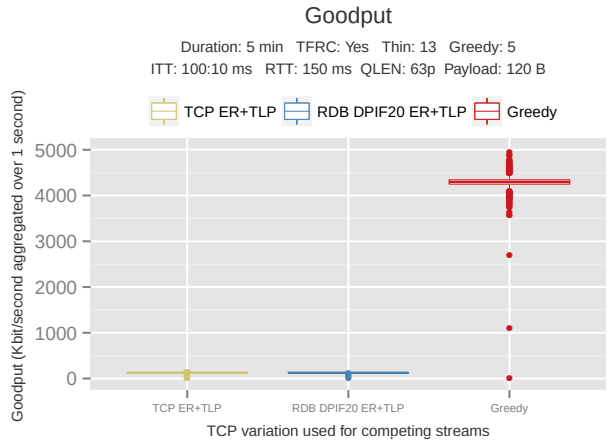
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
247	239	241	244	246	249	251	253	273	383	494	645	778	3263	7739
238	238	241	243	245	248	249	250	250	251	252	327	580	3268	7464
262	241	244	247	258	347	422	461	476	487	498	530	671	777	999

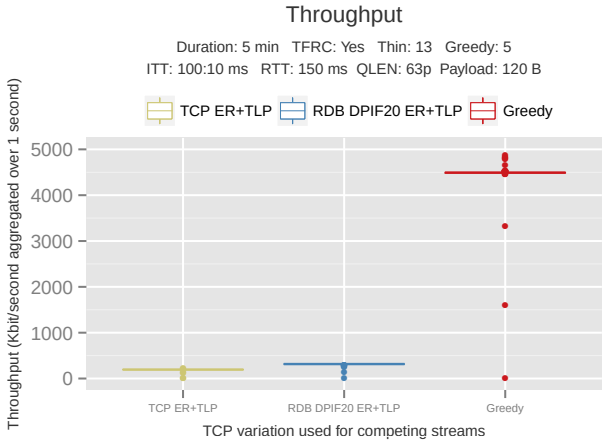
(c)

Figure A.3.74



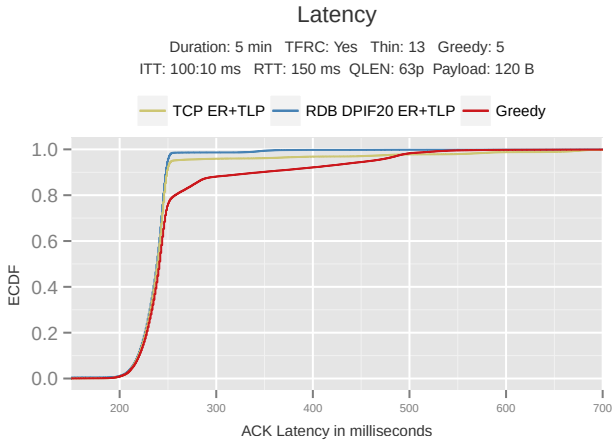
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	98 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF20 ER+TLP	rdbsender	13	99 / 99 / 100	239/240/240	9/1/7/8/9/22
Greedy	zsender	5	12 / 13 / 15	1447/1447/1447	19/3/14/18/22/53

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
39.6 K	419	437	0	4.8 M	4.7 M	51.1 K	51.1 K	1.05 / 1.06	1473
38.8 K	453	49	38.8 K	9.3 M	4.7 M	4.7 M	5.8 K	1.16 / 1.16	117
113.0 K	1149	1151	0	163.5 M	161.8 M	1.7 M	1.7 M	1.02 / 1.02	14116

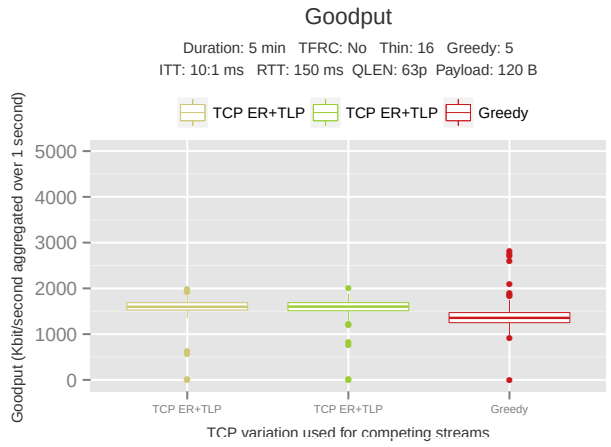
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
248	239	242	244	246	249	251	255	316	447	554	659	780	1151	1151
238	239	241	243	245	248	249	250	250	251	253	340	674	801	890
262	241	244	247	257	345	420	461	475	486	496	523	708	849	1152

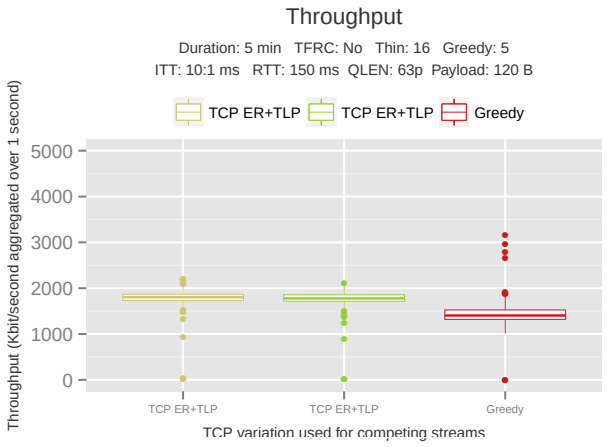
(c)

Figure A.3.75



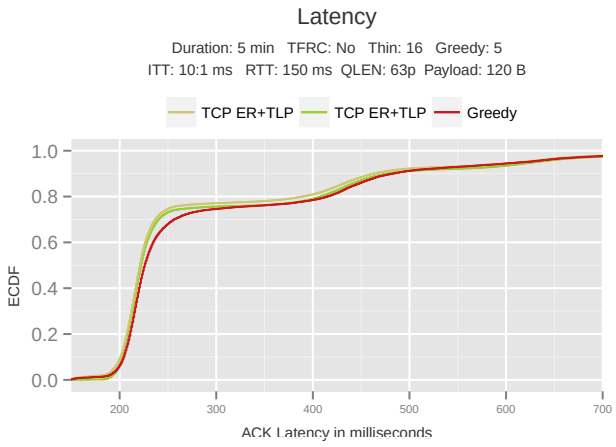
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	38 / 41 / 46	497/538/614	7/1/5/6/8/23
TCP ER+TLP	rdbsender	16	40 / 44 / 47	526/588/632	6/1/4/6/8/21
Greedy	zsender	5	37 / 39 / 41	1445/1446/1447	7/1/4/6/8/29

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
117.4 K	5575	5588	0	63.0 M	60.3 M	2.8 M	2.8 M	4.43 / 4.75	24176
106.4 K	5692	5705	0	62.5 M	59.3 M	3.1 M	3.1 M	5.04 / 5.34	22869
38.0 K	2113	2115	0	55.0 M	51.9 M	3.1 M	3.1 M	5.55 / 5.56	7931

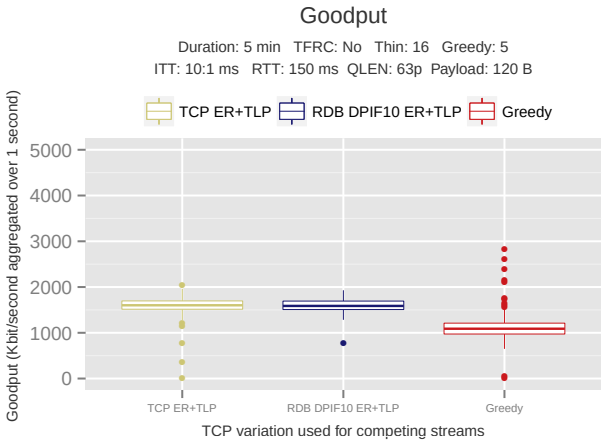
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
287	221	226	237	390	463	544	616	638	662	723	904	1591	3754	4193
296	222	228	240	408	477	587	630	650	679	818	959	1715	3834	4327
298	226	235	256	415	483	551	618	642	670	730	904	1556	5077	5667

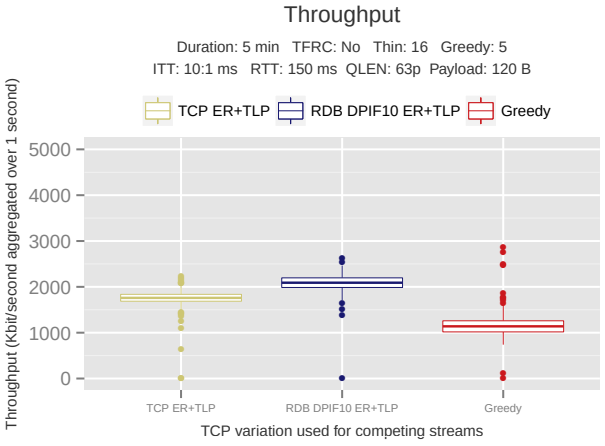
(c)

Figure A.3.76



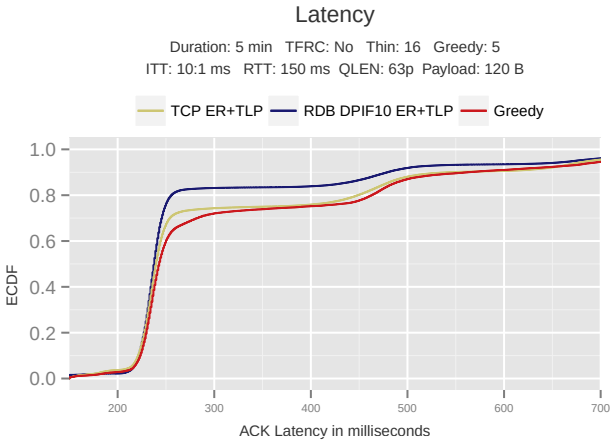
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	47 / 50 / 56	612/661/750	6/1/4/6/8/23
RDB DPIF10 ER+TLP	rdbsender	16	49 / 52 / 55	801/850/874	6/1/4/6/8/26
Greedy	zsender	5	43 / 48 / 53	1445/1446/1447	6/1/4/5/7/25

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
96.2 K	5553	5566	0	63.4 M	60.1 M	3.3 M	3.3 M	5.20 / 5.77	20435
90.7 K	5658	3759	42.6 K	77.0 M	59.6 M	17.5 M	3.3 M	6.07 / 6.24	12113
31.1 K	1939	1941	0	45.0 M	42.2 M	2.8 M	2.8 M	6.23 / 6.23	6603

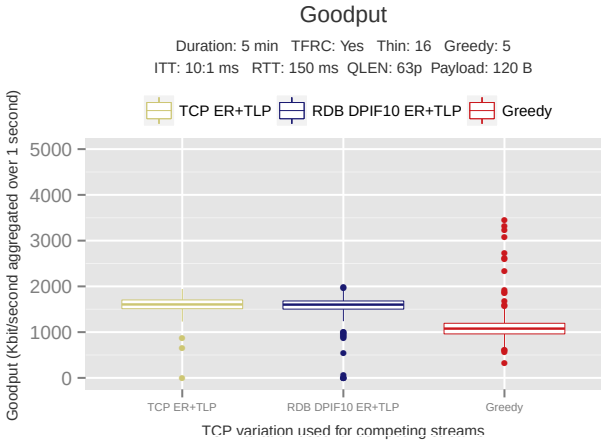
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
327	240	245	254	449	542	663	689	712	768	928	1139	2041	9030	9171
299	238	241	246	256	479	531	674	698	739	916	1113	1830	3235	4795
343	243	250	282	463	562	670	708	729	808	939	1207	4558	6804	8413

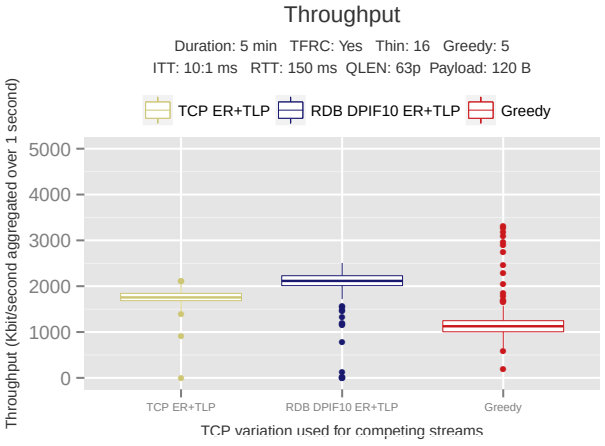
(c)

Figure A.3.77



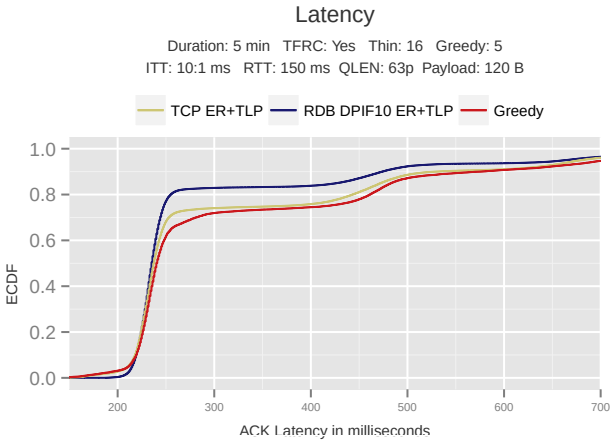
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	46 / 49 / 55	604/654/733	6/1/4/6/7/26
RDB DPIF10 ER+TLP	rdbsender	16	46 / 51 / 55	755/821/876	7/1/4/6/8/132
Greedy	zsender	5	47 / 48 / 49	1445/1446/1447	6/1/4/5/7/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
97.4 K	5576	5576	0	63.6 M	60.3 M	3.3 M	3.3 M	5.18 / 5.72	21206
93.0 K	5849	3640	47.8 K	76.3 M	58.3 M	18.0 M	3.2 M	6.12 / 6.28	12733
31.3 K	2037	2037	0	45.2 M	42.3 M	2.9 M	2.9 M	6.51 / 6.51	6745

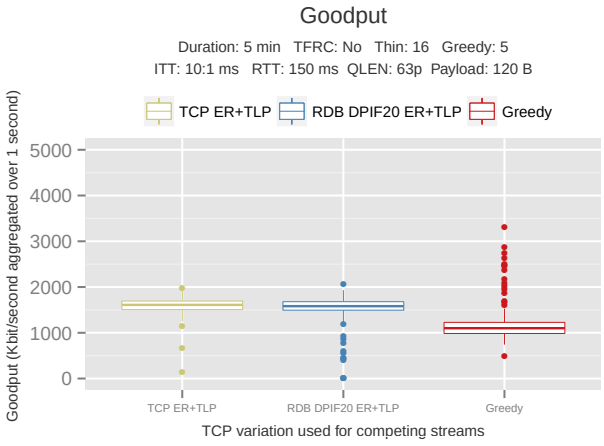
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
320	237	243	253	443	529	651	680	699	732	903	1065	1754	2599	2973
298	235	239	244	255	474	520	663	686	724	908	1119	2060	3820	9689
335	240	248	281	461	575	668	705	725	854	929	1155	2463	5142	6252

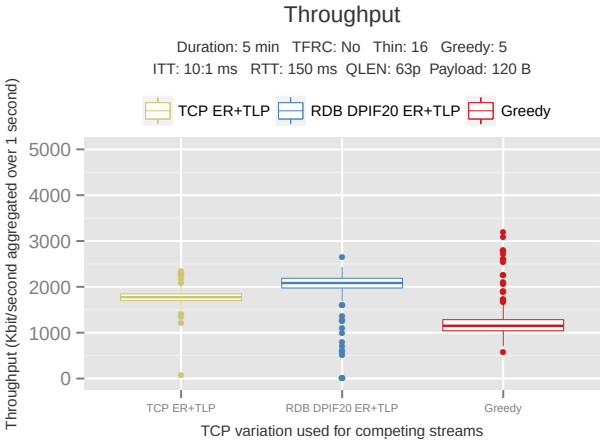
(c)

Figure A.3.78



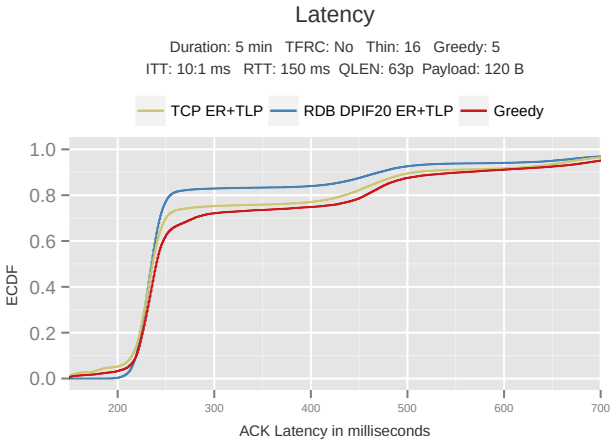
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	44 / 47 / 51	579/626/682	6/1/4/6/8/24
RDB DPIF20 ER+TLP	rdbsender	16	50 / 52 / 55	742/822/855	6/1/4/6/8/19
Greedy	zsender	5	45 / 46 / 48	1445/1446/1447	6/1/4/5/7/29

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
101.6 K	5478	5478	0	63.5 M	60.3 M	3.1 M	3.1 M	4.95 / 5.39	21437
90.4 K	5376	3547	39.8 K	74.3 M	57.9 M	16.4 M	3.0 M	5.87 / 5.94	12486
32.3 K	2059	2059	0	46.8 M	43.8 M	3.0 M	3.0 M	6.35 / 6.37	6949

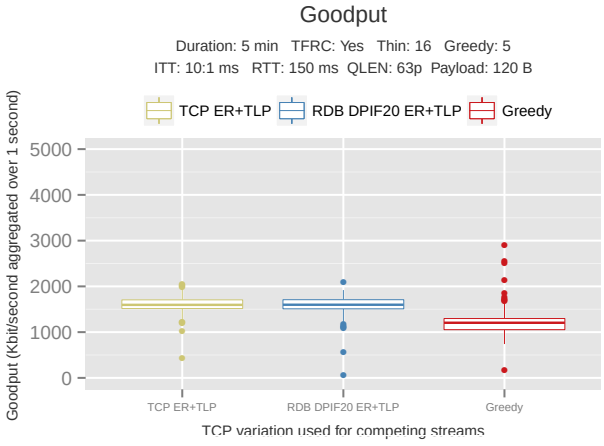
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
312	236	241	250	435	507	642	673	689	717	882	992	1600	2714	4254
296	235	239	244	255	471	508	650	674	705	875	1016	1857	7599	35670
334	240	247	279	457	556	663	698	718	783	924	1130	3187	6473	6481

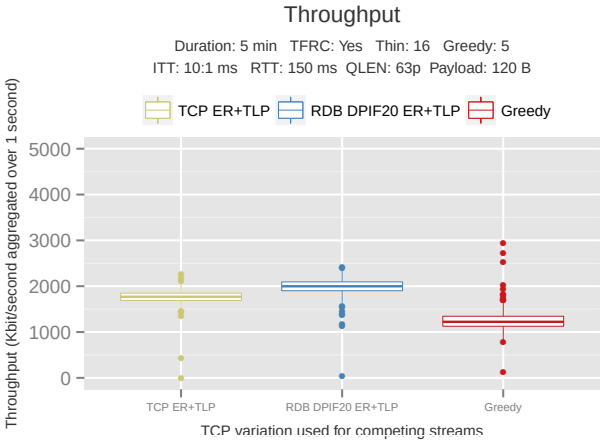
(c)

Figure A.3.79



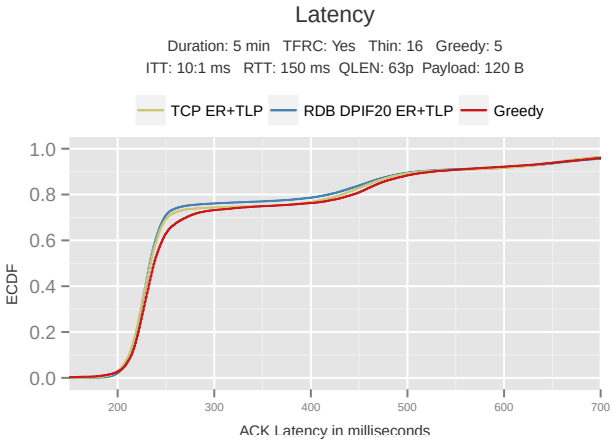
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	44 / 48 / 52	583/637/706	6/1/4/6/8/22
RDB DPIF20 ER+TLP	rdbsender	16	47 / 49 / 52	721/757/806	7/1/4/6/9/27
Greedy	zsender	5	43 / 45 / 47	1446/1447/1447	6/1/4/5/7/33

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
99.8 K	5784	5784	0	63.5 M	60.2 M	3.3 M	3.3 M	5.21 / 5.79	21090
96.1 K	6291	5084	23.9 K	72.7 M	59.6 M	13.1 M	3.7 M	6.47 / 6.54	18907
33.5 K	2088	2088	0	48.4 M	45.4 M	3.0 M	3.0 M	6.24 / 6.24	6827

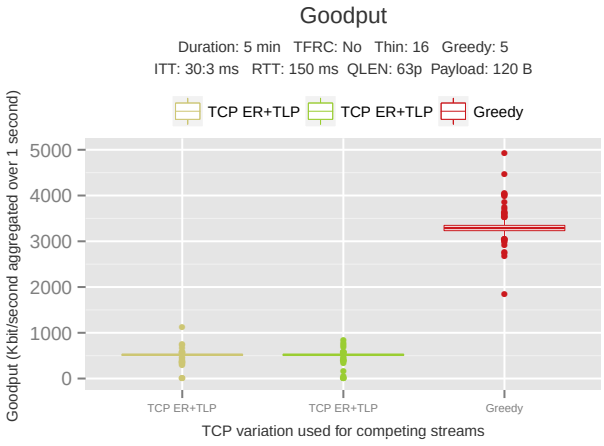
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
314	234	240	252	432	514	634	669	689	723	892	1093	1757	3048	4013
316	233	239	248	418	510	637	681	717	872	969	1161	2170	2938	3887
321	238	246	271	444	527	629	677	701	745	896	1093	2135	3630	4389

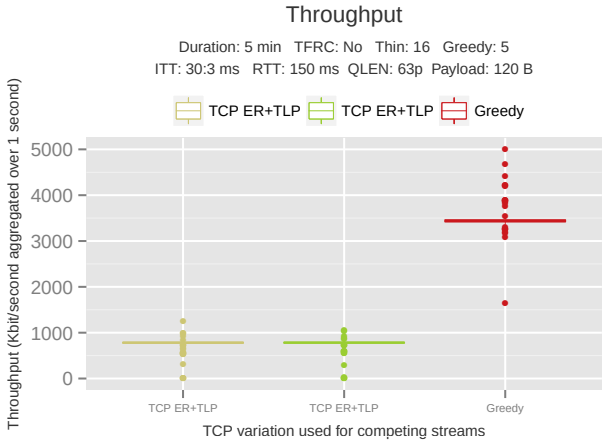
(c)

Figure A.3.80



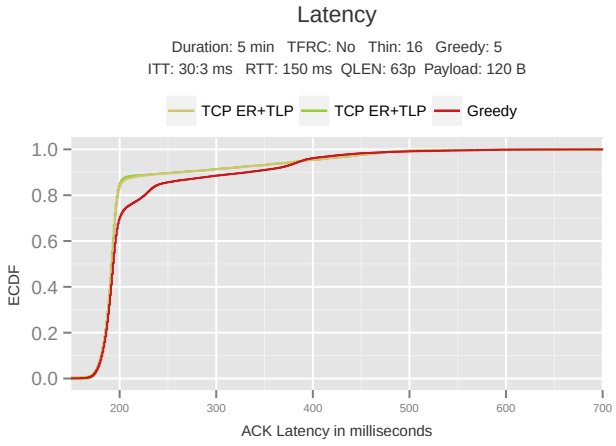
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	32 / 32 / 34	129/132/138	9/1/7/10/11/13
TCP ER+TLP	rdbsender	16	32 / 32 / 33	129/132/136	9/1/7/10/11/13
Greedy	zsender	5	16 / 17 / 18	1447/1447/1448	12/4/9/11/13/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
149.4 K	2339	2352	0	19.7 M	19.4 M	301.7 K	301.7 K	1.53 / 1.56	15845
146.4 K	2310	2312	0	19.3 M	19.0 M	291.6 K	291.6 K	1.51 / 1.58	15316
88.0 K	1565	1565	0	127.3 M	125.0 M	2.3 M	2.3 M	1.78 / 1.78	12049

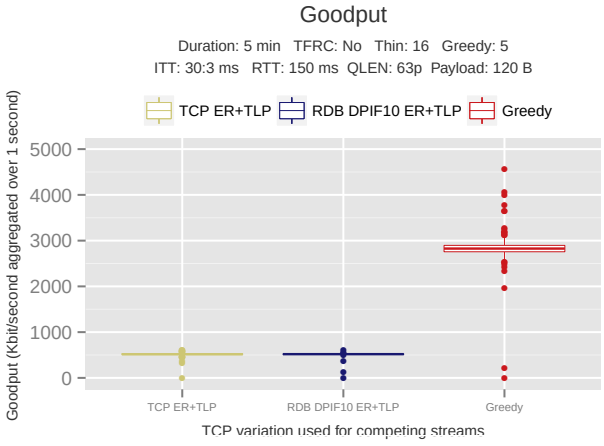
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
212	192	193	195	198	264	347	390	414	439	460	481	673	3259	16500
212	192	193	195	198	260	345	388	413	438	460	481	690	3261	10930
220	193	196	200	226	332	375	389	398	417	441	490	614	745	999

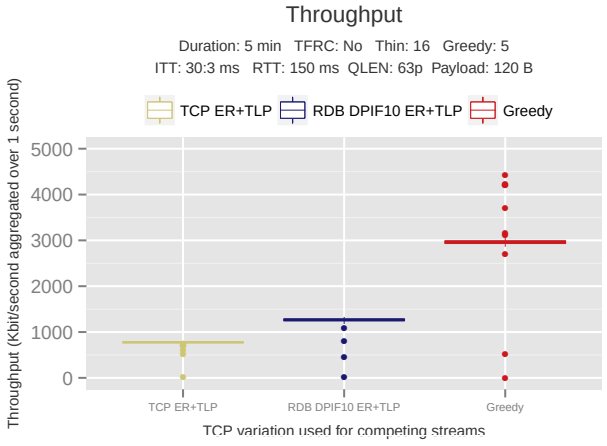
(c)

Figure A.3.81



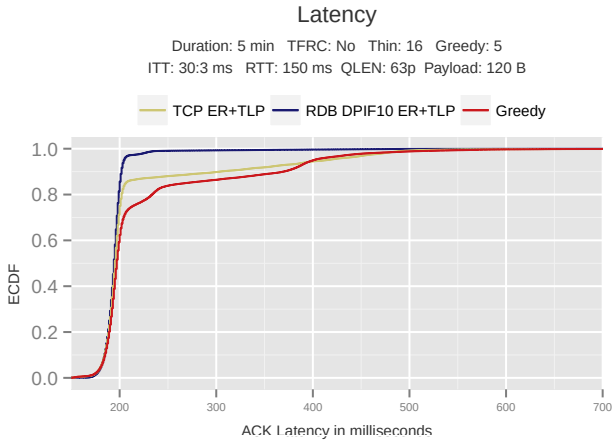
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	32 / 32 / 33	131/134/138	9/1/7/9/11/12
RDB DPIF10 ER+TLP	rdbsender	16	33 / 34 / 34	264/273/280	8/1/6/9/11/12
Greedy	zsender	5	19 / 20 / 21	1447/1447/1447	10/3/8/10/11/29

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
148.7 K	2711	2724	0	19.9 M	19.5 M	346.9 K	346.9 K	1.75 / 1.82	17714
140.6 K	2710	189	140.4 K	38.4 M	19.2 M	19.2 M	24.5 K	1.93 / 1.93	941
75.8 K	1714	1716	0	109.8 M	107.3 M	2.5 M	2.5 M	2.26 / 2.26	11542

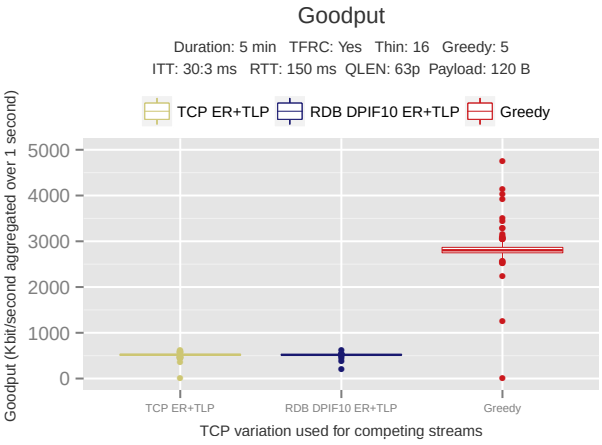
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
218	195	197	199	202	304	373	414	433	452	473	491	711	1135	1268
196	194	196	197	199	202	203	205	206	209	225	239	497	999	1152
228	197	200	204	234	370	389	401	412	431	457	514	743	1023	1152

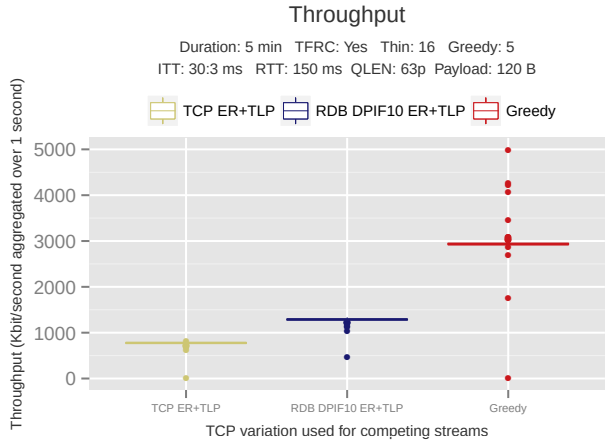
(c)

Figure A.3.82



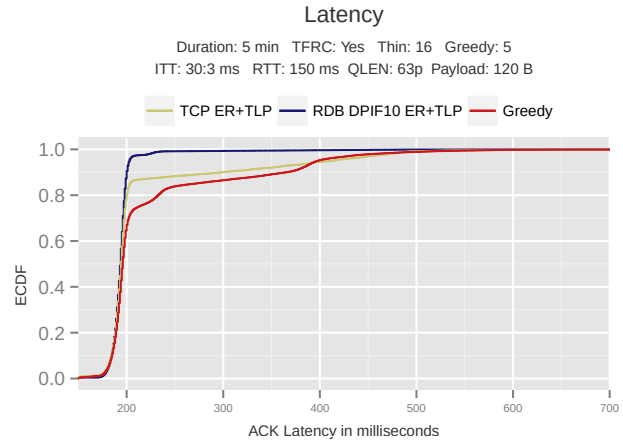
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	32 / 32 / 33	132/134/137	9/1/7/9/11/12
RDB DPIF10 ER+TLP	rdbsender	16	30 / 31 / 31	244/251/255	9/1/8/10/11/27
Greedy	zsender	5	19 / 20 / 20	1446/1447/1447	10/1/8/9/11/33

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
148.5 K	2670	2670	0	19.9 M	19.5 M	348.4 K	348.4 K	1.75 / 1.80	17408
153.6 K	2884	188	153.3 K	38.5 M	19.3 M	19.3 M	22.9 K	1.86 / 1.88	1071
75.5 K	1704	1706	0	109.2 M	106.7 M	2.5 M	2.5 M	2.26 / 2.26	11459

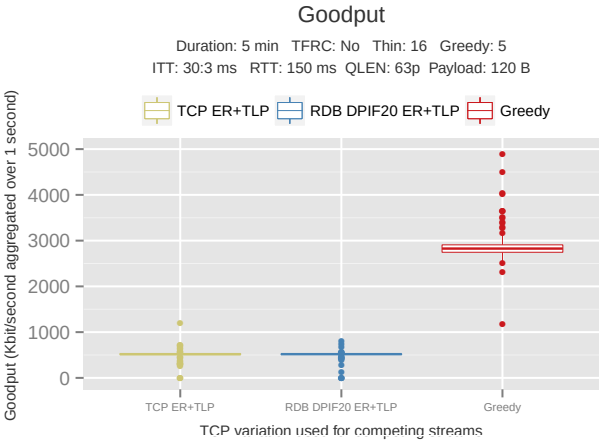
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
217	194	196	198	200	299	368	412	430	451	473	493	722	961	1408
195	194	195	196	198	200	202	203	204	207	224	234	497	1053	1262
227	196	198	202	233	365	387	398	409	428	455	507	727	1585	1744

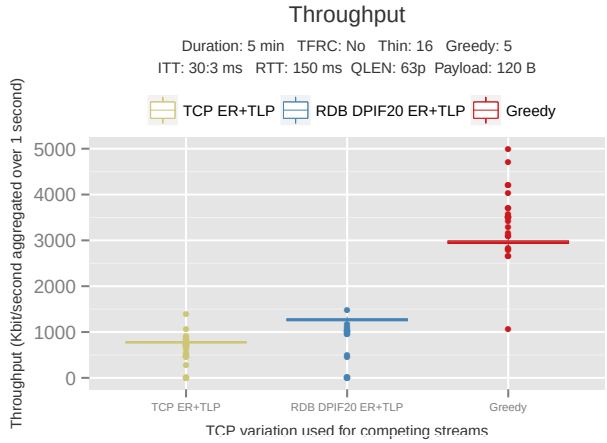
(c)

Figure A.3.83



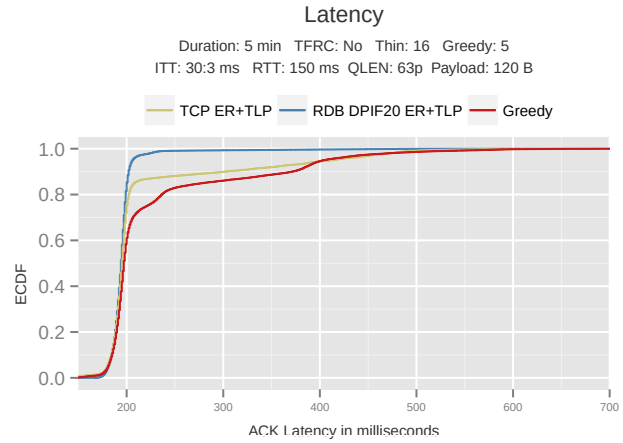
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	32 / 33 / 34	132/135/141	9/1/7/9/11/13
RDB DPIF20 ER+TLP	rdbsender	16	33 / 34 / 36	263/270/280	8/1/6/9/11/13
Greedy	zsender	5	17 / 19 / 20	1447/1447/1447	11/2/8/10/12/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
146.3 K	2594	2607	0	19.8 M	19.5 M	340.5 K	340.5 K	1.72 / 1.77	17313
139.8 K	2594	180	138.8 K	37.8 M	19.0 M	18.8 M	25.6 K	1.85 / 1.85	964
76.5 K	1688	1688	0	110.7 M	108.2 M	2.4 M	2.4 M	2.21 / 2.21	11843

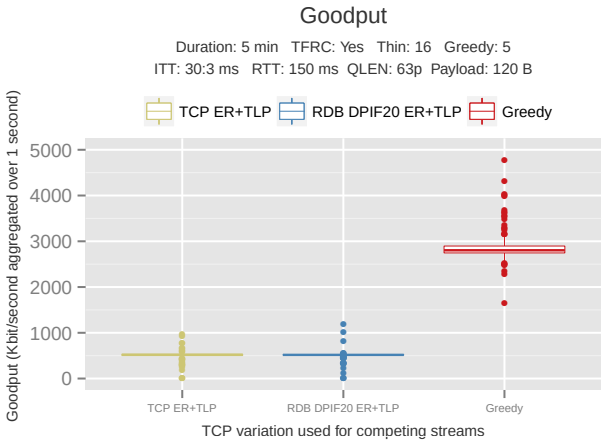
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
218	195	197	199	202	302	372	413	432	452	473	494	686	3219	7689
197	194	196	197	199	202	204	206	208	213	224	239	486	3148	10090
229	197	200	207	236	371	390	404	420	439	471	528	654	813	999

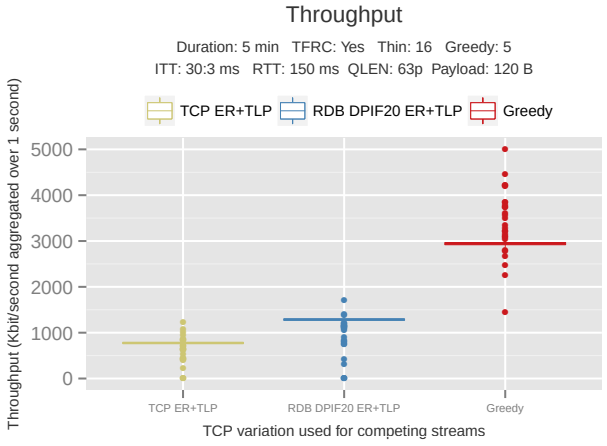
(c)

Figure A.3.84



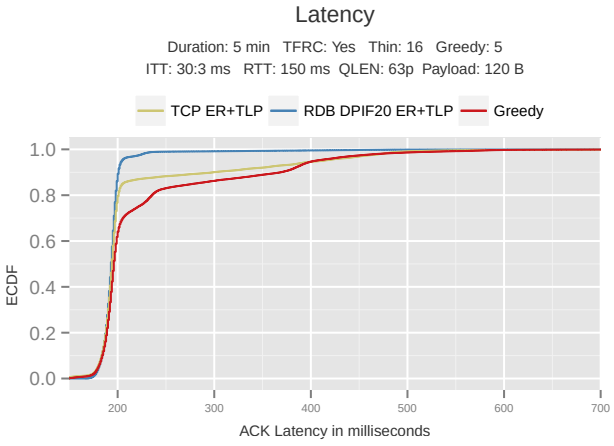
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	32 / 33 / 35	132/136/143	9/1/7/9/11/12
RDB DPIF20 ER+TLP	rdbsender	16	31 / 32 / 33	248/251/256	10/1/8/10/11/119
Greedy	zsender	5	18 / 19 / 20	1447/1447/1447	10/2/8/10/12/44

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
145.2 K	2600	2604	0	19.8 M	19.4 M	343.0 K	343.0 K	1.74 / 1.79	17002
150.2 K	2794	238	149.1 K	37.7 M	19.0 M	18.7 M	31.0 K	1.85 / 1.86	1271
76.2 K	1711	1711	0	110.3 M	107.8 M	2.5 M	2.5 M	2.25 / 2.24	11736

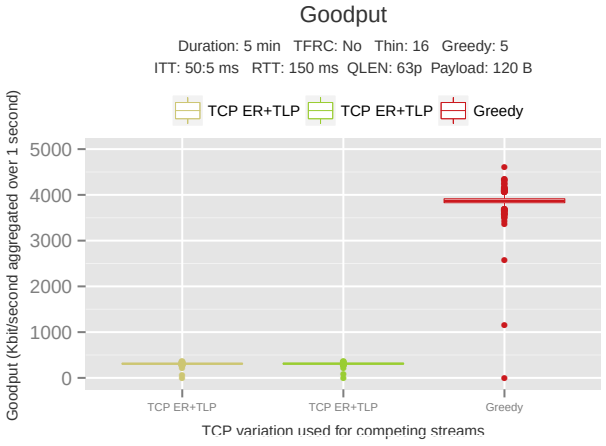
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
218	194	196	198	201	299	370	411	430	451	473	495	734	6029	17200
197	193	195	196	198	201	202	204	207	217	226	254	513	3243	17190
229	196	199	206	236	369	389	404	422	441	469	528	695	873	999

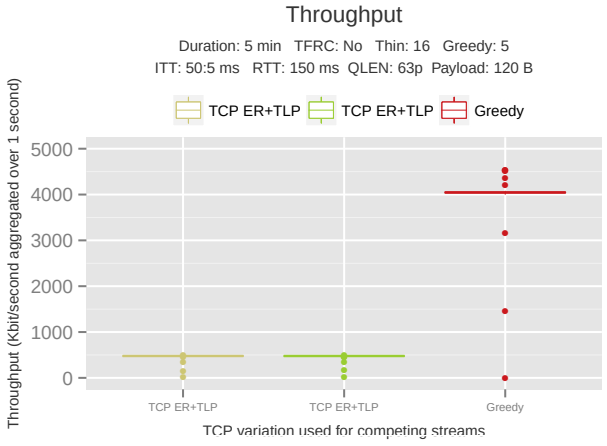
(c)

Figure A.3.85



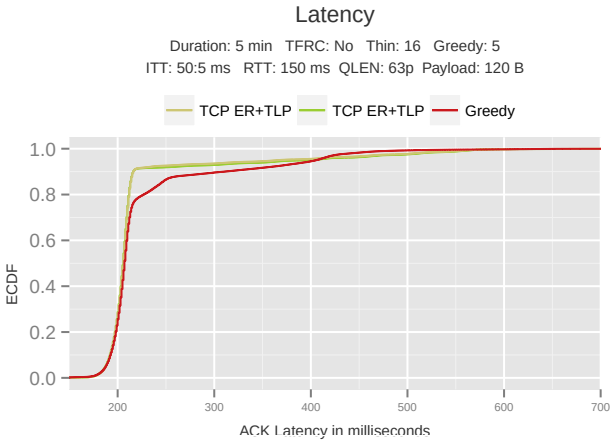
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	50 / 50 / 50	122/122/123	8/1/7/9/9/10
TCP ER+TLP	rdbsender	16	50 / 50 / 51	122/123/124	8/1/7/9/9/10
Greedy	zsender	5	14 / 14 / 15	1446/1447/1447	14/3/11/14/17/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
96.3 K	1204	1217	0	11.8 M	11.6 M	145.3 K	145.3 K	1.23 / 1.25	6711
94.7 K	1311	1325	0	11.6 M	11.5 M	158.2 K	158.2 K	1.36 / 1.38	7185
102.2 K	1361	1363	0	147.9 M	145.9 M	2.0 M	2.0 M	1.33 / 1.33	12357

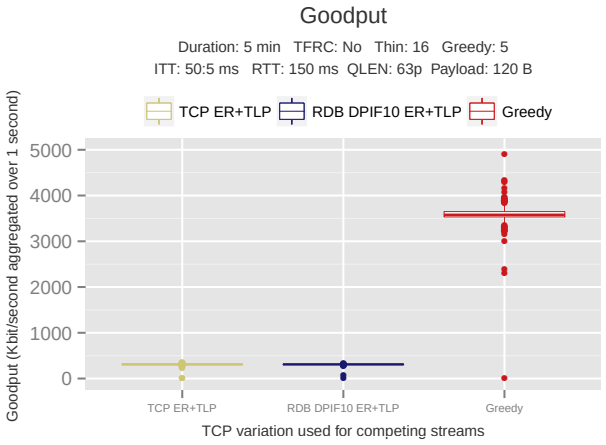
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
221	206	208	210	212	216	266	366	413	461	507	552	653	1152	1153
222	206	208	209	212	216	304	397	427	470	514	556	681	1151	1152
229	207	210	212	227	311	376	405	414	422	442	477	646	849	1151

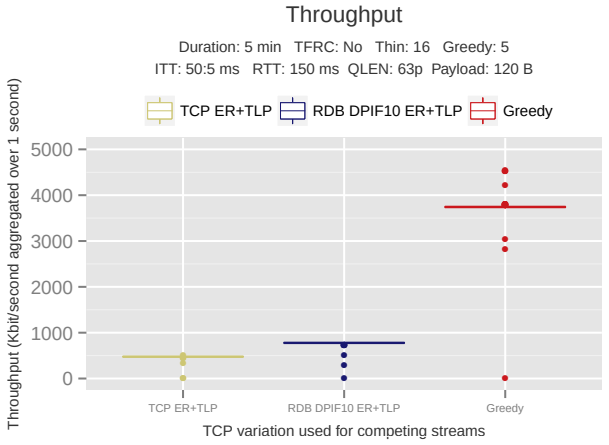
(c)

Figure A.3.86



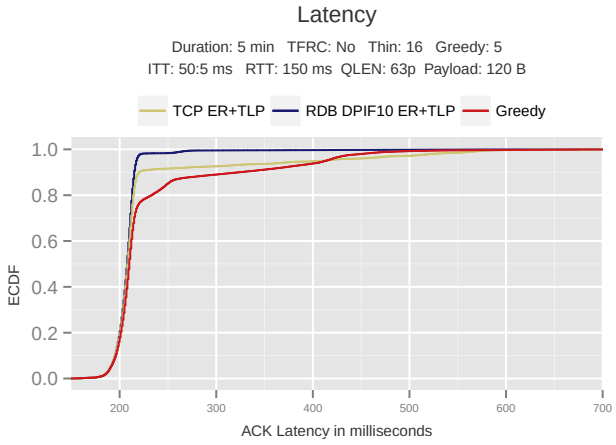
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	50 / 50 / 51	122/123/124	8/1/7/9/9/10
RDB DPIF10 ER+TLP	rdbsender	16	50 / 51 / 51	243/244/248	8/1/7/9/9/10
Greedy	zsender	5	15 / 16 / 16	1446/1447/1447	14/4/11/13/16/40

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
96.0 K	1380	1393	0	11.8 M	11.6 M	168.1 K	168.1 K	1.42 / 1.44	7475
93.8 K	1270	91	93.7 K	22.9 M	11.5 M	11.5 M	9.7 K	1.35 / 1.35	358
95.0 K	1360	1362	0	137.5 M	135.6 M	2.0 M	2.0 M	1.43 / 1.43	11920

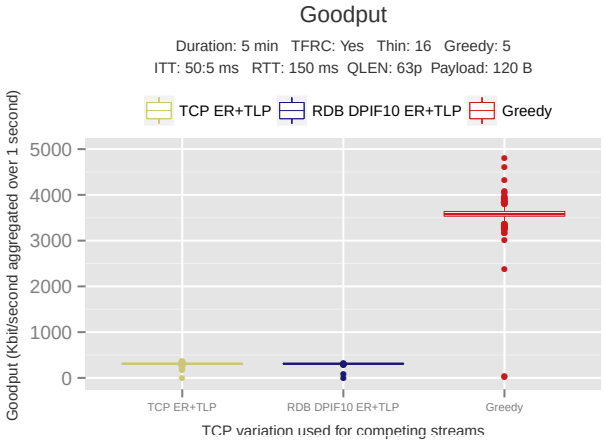
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
226	209	210	212	214	220	316	410	449	484	523	566	728	1152	1337
209	208	210	211	213	215	216	217	218	219	222	263	558	1151	1152
233	210	212	215	233	324	386	413	421	430	450	483	672	1005	1152

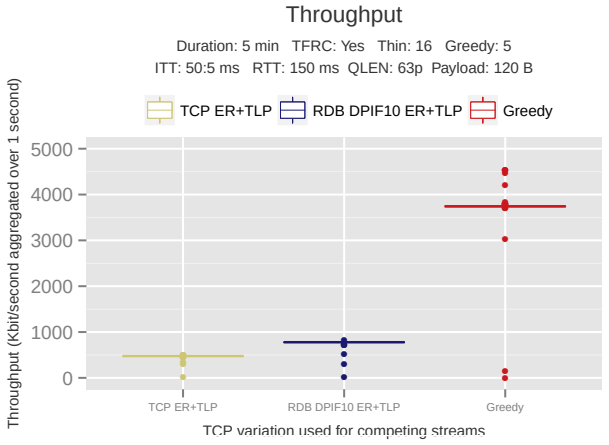
(c)

Figure A.3.87



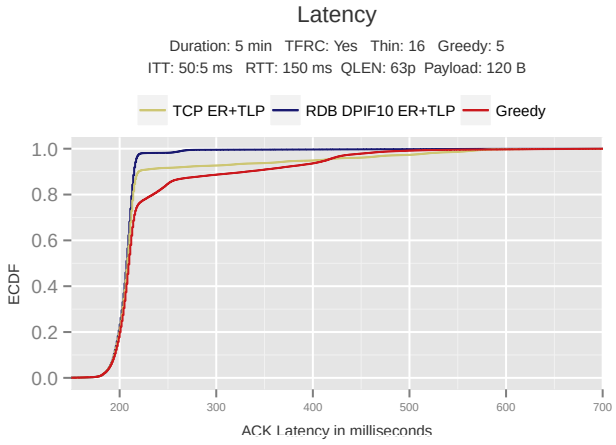
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	50 / 50 / 50	122/123/124	8/1/7/9/9/10
RDB DPIF10 ER+TLP	rdbsender	16	50 / 50 / 50	240/241/241	10/1/9/9/9/90
Greedy	zsender	5	15 / 16 / 16	1447/1447/1448	14/3/11/13/16/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
96.0 K	1366	1379	0	11.8 M	11.6 M	166.4 K	166.4 K	1.41 / 1.42	7441
95.2 K	1388	91	95.0 K	22.9 M	11.5 M	11.5 M	9.4 K	1.46 / 1.46	403
94.7 K	1365	1367	0	137.1 M	135.2 M	2.0 M	2.0 M	1.44 / 1.44	12276

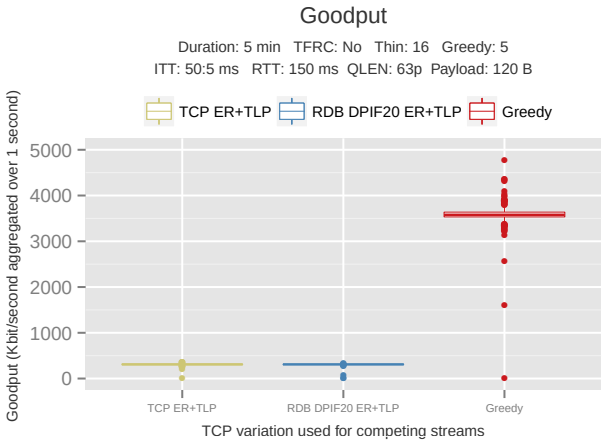
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
225	208	210	212	214	220	315	406	441	477	520	561	648	1152	1250
208	207	209	211	213	215	216	217	218	219	223	264	562	1152	1235
234	210	212	215	234	331	390	414	421	431	454	490	669	827	1150

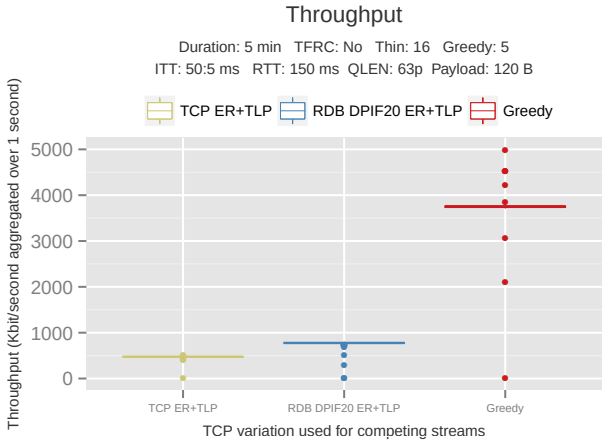
(c)

Figure A.3.88



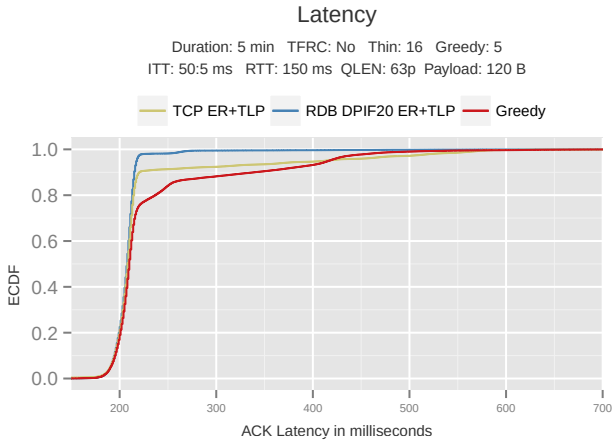
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	50 / 50 / 50	122/123/124	8/1/7/9/9/10
RDB DPIF20 ER+TLP	rdbsender	16	50 / 51 / 51	243/245/247	8/1/6/9/9/10
Greedy	zsender	5	15 / 16 / 17	1447/1447/1448	14/3/11/13/16/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
95.9 K	1413	1413	0	11.8 M	11.6 M	170.5 K	170.5 K	1.44 / 1.47	7735
93.5 K	1380	104	93.3 K	22.9 M	11.5 M	11.4 M	10.9 K	1.48 / 1.47	410
95.1 K	1430	1432	0	137.7 M	135.6 M	2.1 M	2.1 M	1.50 / 1.50	12721

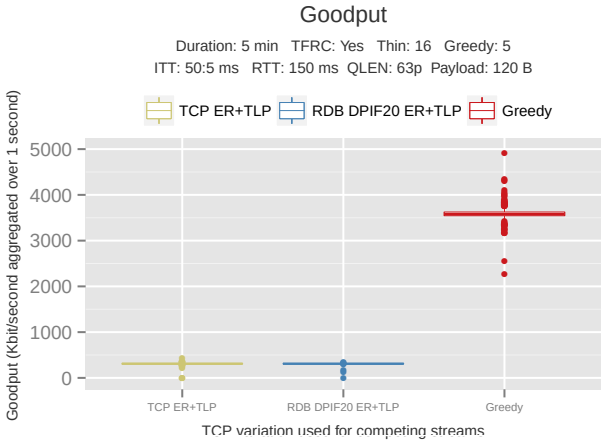
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
226	208	210	212	215	221	322	412	453	484	523	563	684	935	1099
209	208	209	211	213	215	217	218	218	220	226	265	564	1151	1153
235	210	212	216	237	340	397	417	424	434	456	498	671	835	1152

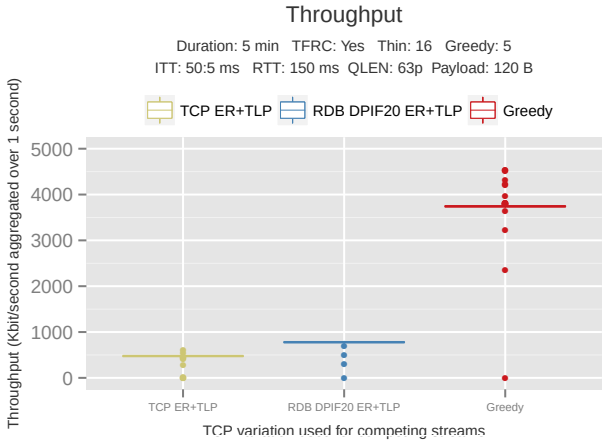
(c)

Figure A.3.89



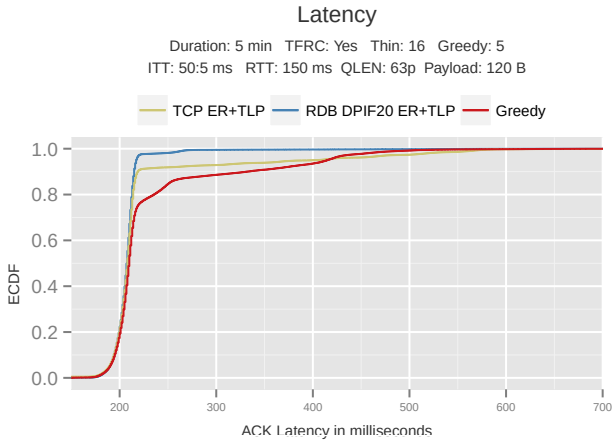
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	50 / 51 / 52	122/123/126	8/1/7/9/9/10
RDB DPIF20 ER+TLP	rdbsender	16	50 / 50 / 50	240/241/241	9/1/9/9/10/22
Greedy	zsender	5	15 / 16 / 16	1447/1447/1447	13/4/11/13/15/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
95.6 K	1326	1333	0	11.8 M	11.6 M	159.0 K	159.0 K	1.35 / 1.39	7224
95.5 K	1350	86	95.2 K	23.0 M	11.5 M	11.5 M	10.2 K	1.41 / 1.41	430
95.0 K	1385	1385	0	137.5 M	135.5 M	2.0 M	2.0 M	1.46 / 1.46	12357

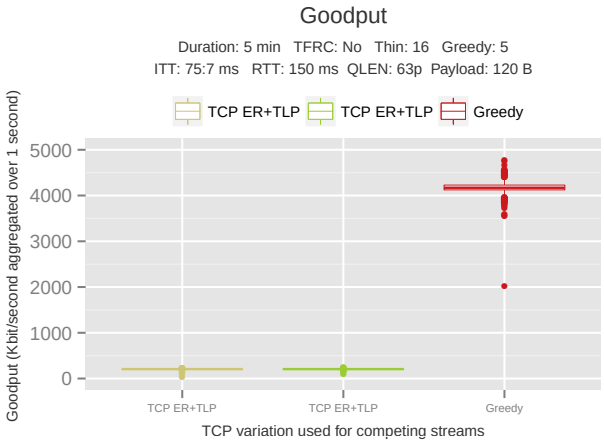
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
225	208	210	212	214	219	308	403	430	474	518	562	691	2368	7621
208	208	209	211	213	215	216	217	218	220	243	264	559	950	1297
234	210	212	215	236	332	391	415	422	431	456	488	646	764	999

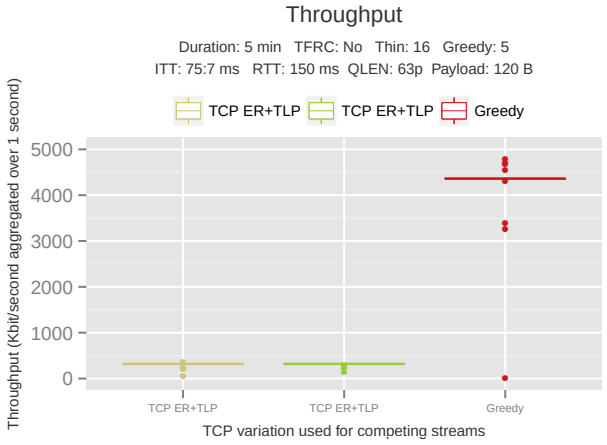
(c)

Figure A.3.90



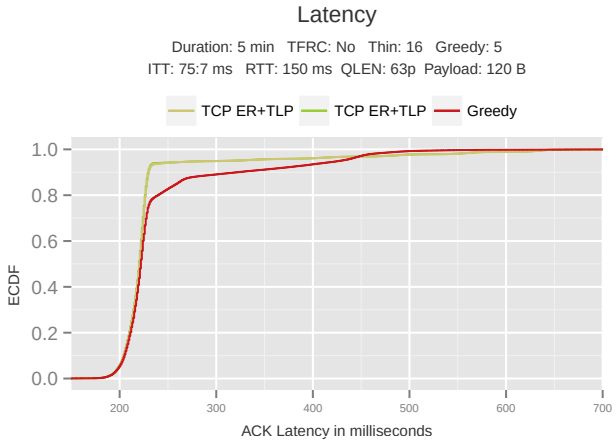
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 74 / 74	120/120/120	7/1/7/8/8/10
TCP ER+TLP	rdbsender	16	74 / 74 / 74	120/120/121	7/1/7/8/8/10
Greedy	zsender	5	13 / 13 / 14	1446/1447/1447	17/6/13/16/19/39

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
65.1 K	717	720	0	7.8 M	7.7 M	86.5 K	86.5 K	1.10 / 1.10	3256
64.4 K	711	711	0	7.7 M	7.7 M	85.4 K	85.4 K	1.10 / 1.10	3186
110.0 K	1235	1235	0	159.2 M	157.4 M	1.8 M	1.8 M	1.12 / 1.12	13614

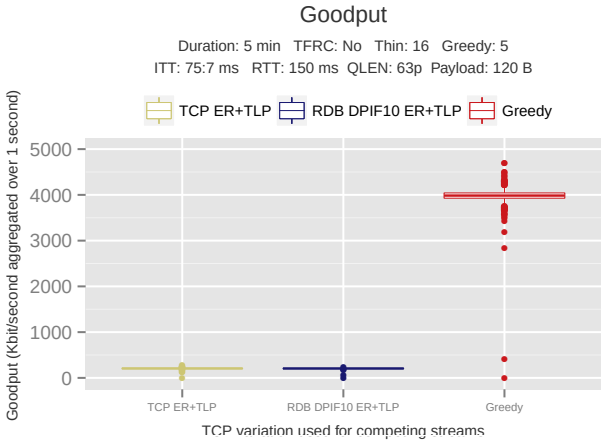
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
232	220	223	225	227	230	233	316	394	470	544	617	652	808	940
232	220	222	224	226	230	232	310	391	467	543	616	653	890	1186
243	222	224	228	240	322	391	428	440	449	461	489	664	898	999

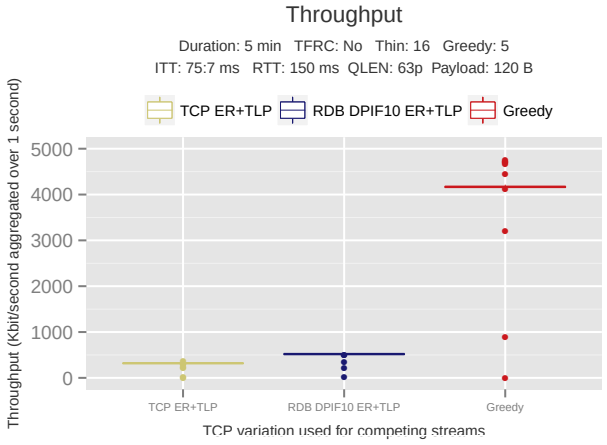
(c)

Figure A.3.91



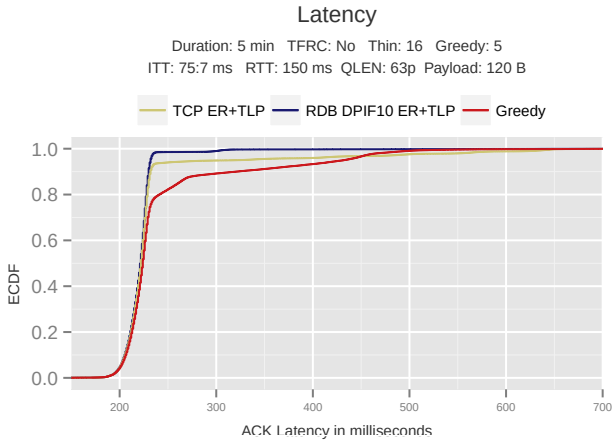
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 74 / 74	120/120/120	7/1/7/8/8/10
RDB DPIF10 ER+TLP	rdbsender	16	75 / 75 / 75	240/240/241	7/1/7/8/8/10
Greedy	zsender	5	14 / 14 / 15	1446/1447/1447	16/1/12/15/18/39

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
65.0 K	738	753	0	7.8 M	7.7 M	88.9 K	88.9 K	1.13 / 1.13	3207
63.3 K	762	65	63.3 K	15.2 M	7.6 M	7.6 M	6.2 K	1.20 / 1.20	155
104.9 K	1181	1183	0	151.8 M	150.1 M	1.7 M	1.7 M	1.12 / 1.13	12620

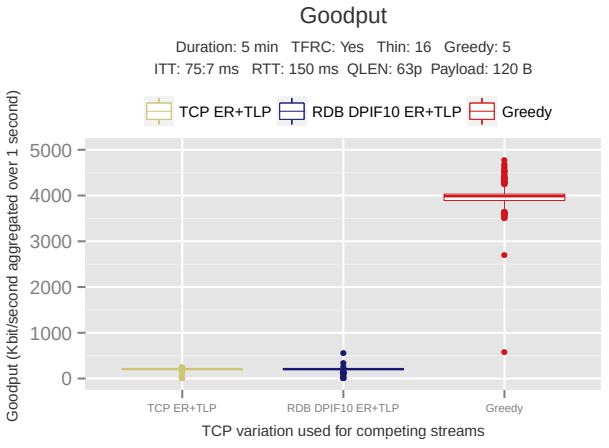
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
235	222	224	227	229	232	235	321	402	477	552	625	680	1152	1152
222	222	224	226	228	230	231	232	233	234	235	300	621	1151	1152
246	224	227	229	241	321	393	432	444	453	467	497	686	2086	2106

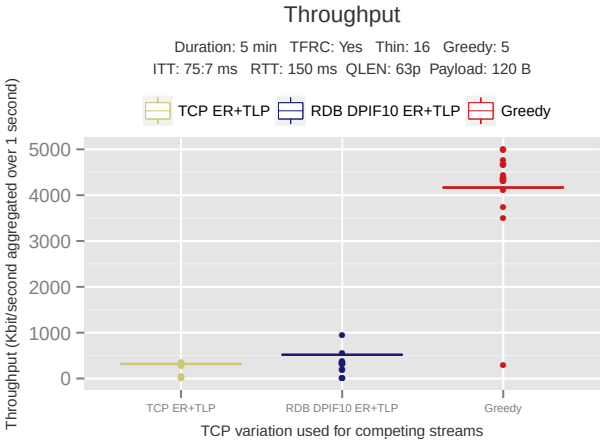
(c)

Figure A.3.92



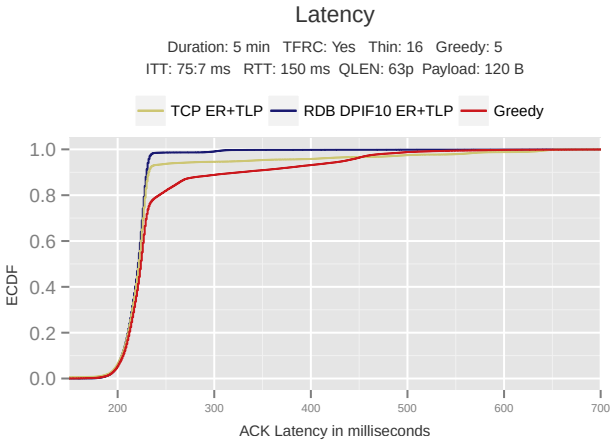
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	75 / 75 / 75	120/120/121	7/1/7/8/8/10
RDB DPIF10 ER+TLP	rdbsender	16	75 / 76 / 78	240/241/244	11/1/8/8/10/104
Greedy	zsender	5	13 / 14 / 15	1447/1447/1447	16/6/13/15/18/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
64.7 K	773	779	0	7.8 M	7.7 M	90.8 K	90.8 K	1.17 / 1.19	3382
62.4 K	727	66	62.3 K	15.1 M	7.6 M	7.5 M	10.2 K	1.17 / 1.16	120
105.1 K	1167	1167	0	152.0 M	150.3 M	1.7 M	1.7 M	1.11 / 1.11	12749

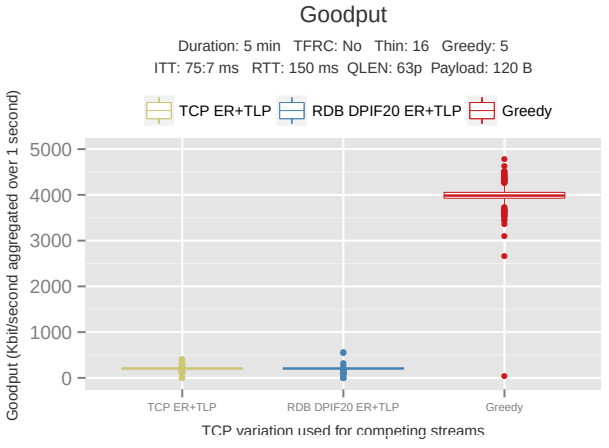
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
235	222	224	227	229	232	237	334	408	481	554	626	659	3261	3747
222	221	224	226	228	230	231	232	233	233	235	298	621	3266	7725
246	224	227	230	243	326	397	435	446	456	474	510	673	867	999

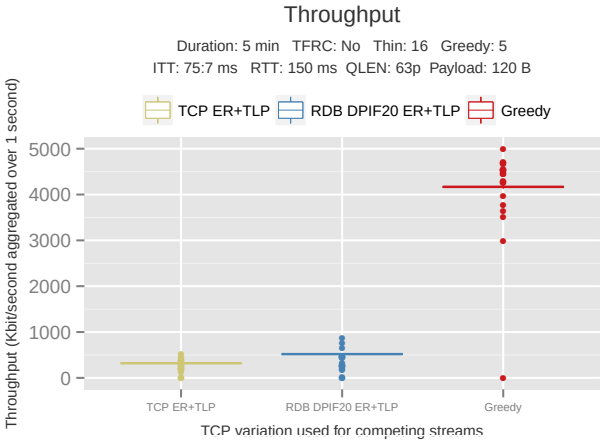
(c)

Figure A.3.93



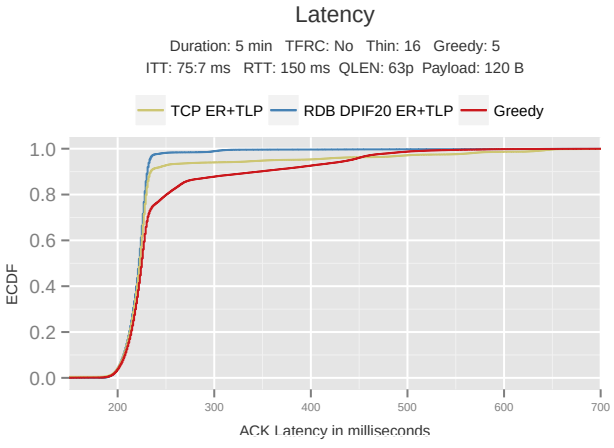
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 76 / 79	120/122/127	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	16	75 / 77 / 80	240/243/248	7/1/7/8/8/10
Greedy	zsender	5	14 / 14 / 15	1447/1447/1447	16/5/13/16/19/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
64.0 K	850	855	0	7.8 M	7.7 M	109.0 K	109.0 K	1.39 / 1.33	3652
61.8 K	802	86	61.6 K	15.0 M	7.6 M	7.4 M	14.3 K	1.30 / 1.30	199
105.5 K	1287	1287	0	152.7 M	150.8 M	1.9 M	1.9 M	1.22 / 1.22	13731

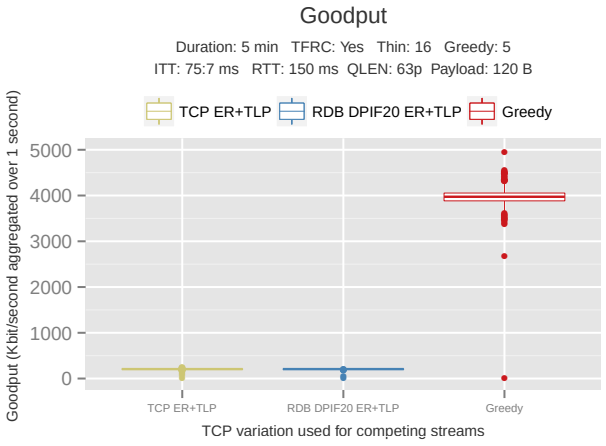
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
240	223	225	227	230	234	252	361	427	496	565	633	723	7646	16990
225	222	224	226	228	231	232	233	234	236	246	302	644	7719	16950
249	225	228	231	251	346	407	440	450	460	480	511	655	779	999

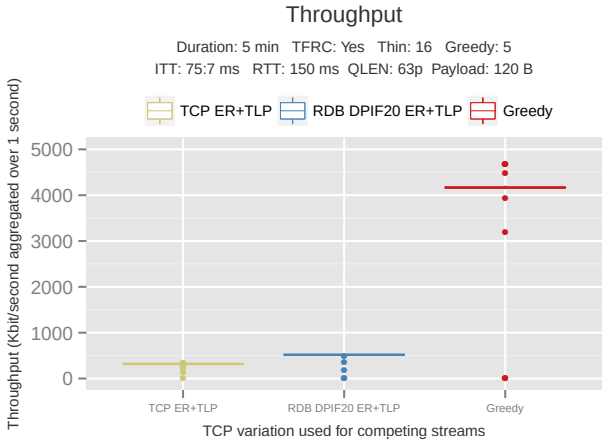
(c)

Figure A.3.94



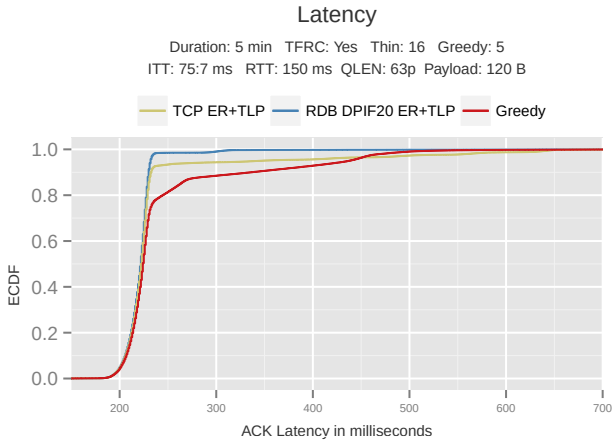
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	74 / 74 / 74	120/120/121	7/1/7/8/8/10
RDB DPIF20 ER+TLP	rdbsender	16	75 / 75 / 75	240/240/240	11/1/8/8/10/97
Greedy	zsender	5	13 / 14 / 17	1446/1447/1447	16/4/13/15/18/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
65.0 K	785	801	0	7.8 M	7.7 M	94.6 K	94.6 K	1.20 / 1.21	3514
63.5 K	804	56	63.5 K	15.3 M	7.6 M	7.6 M	4.4 K	1.26 / 1.26	142
104.8 K	1275	1277	0	151.7 M	149.9 M	1.8 M	1.8 M	1.21 / 1.22	13195

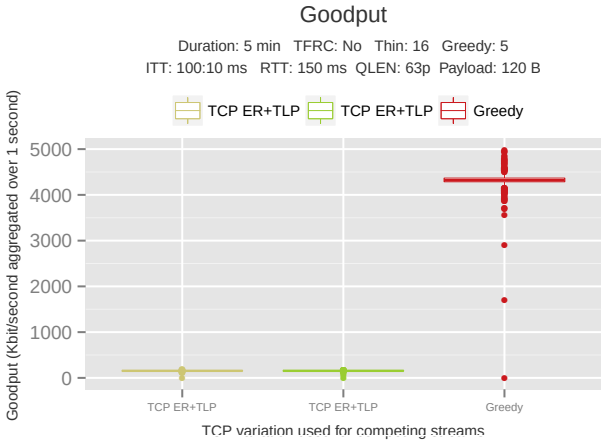
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
236	223	225	227	229	233	243	345	418	488	560	632	726	1151	1210
222	222	224	226	228	230	231	232	233	234	235	299	581	1151	1152
247	224	227	230	245	336	403	438	447	455	470	499	683	999	1152

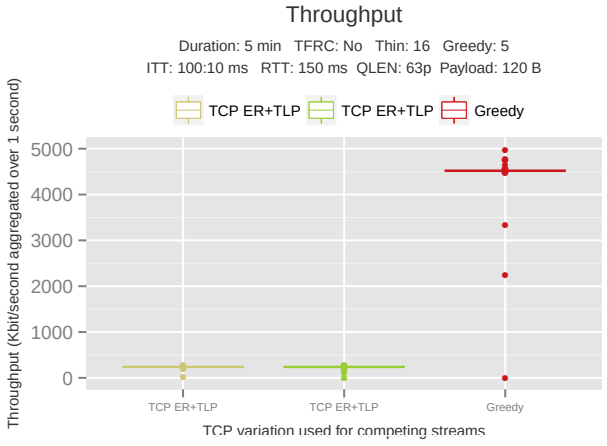
(c)

Figure A.3.95



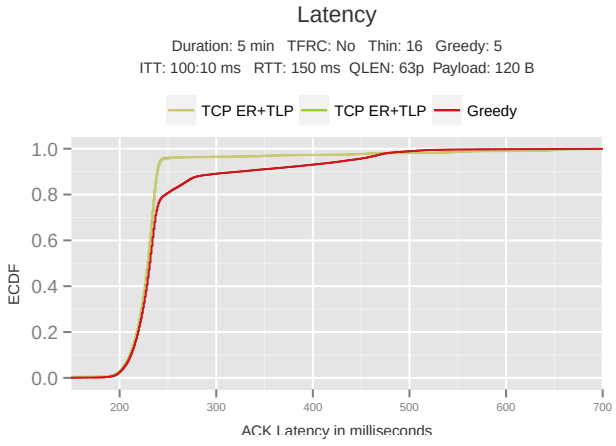
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	98 / 99 / 99	120/120/120	7/1/7/7/7/10
TCP ER+TLP	rdbsender	16	98 / 99 / 99	120/120/120	7/1/7/7/7/10
Greedy	zsender	5	12 / 13 / 14	1446/1447/1447	18/5/14/17/21/48

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.7 K	464	480	0	5.9 M	5.8 M	57.6 K	57.6 K	0.95 / 0.95	1640
48.1 K	446	464	0	5.8 M	5.7 M	55.4 K	55.4 K	0.92 / 0.93	1596
113.8 K	1141	1142	0	164.7 M	163.0 M	1.6 M	1.6 M	1.00 / 1.00	13490

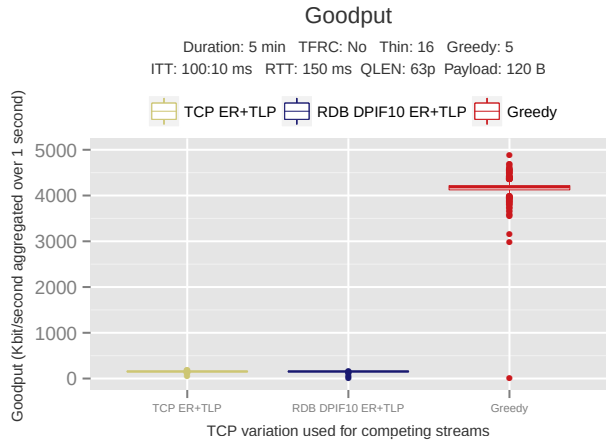
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
237	229	232	234	236	239	241	244	257	362	471	620	749	948	1113
237	229	232	234	236	239	241	243	248	358	467	604	749	942	1074
252	231	234	237	248	324	399	438	454	465	476	504	674	919	1153

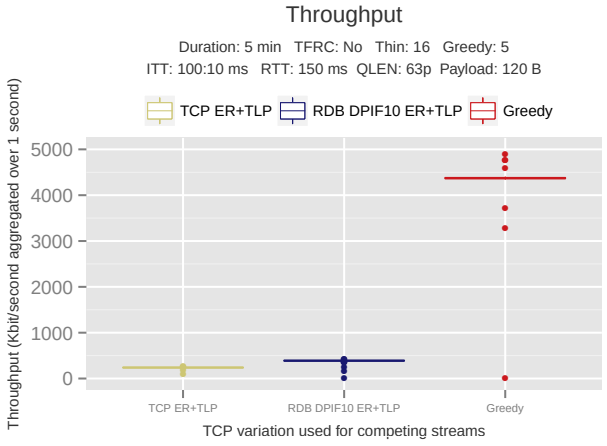
(c)

Figure A.3.96



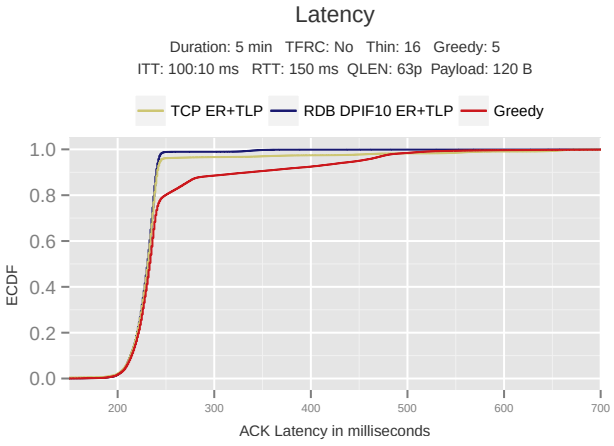
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	98 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF10 ER+TLP	rdbsender	16	100 / 100 / 100	240/240/240	7/1/7/7/7/10
Greedy	zsender	5	12 / 13 / 15	1446/1447/1447	17/5/14/17/20/3

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.6 K	428	446	0	5.8 M	5.8 M	53.5 K	53.5 K	0.88 / 0.88	1558
47.6 K	451	35	47.5 K	11.4 M	5.7 M	5.7 M	2.6 K	0.95 / 0.95	60
109.9 K	1161	1163	0	159.0 M	157.3 M	1.7 M	1.7 M	1.06 / 1.06	13354

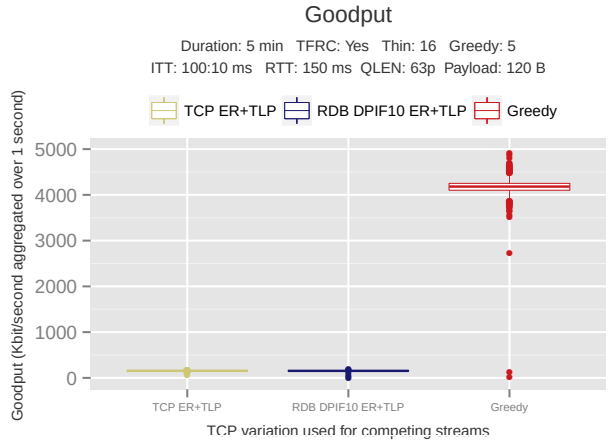
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
238	231	234	236	238	241	243	245	248	352	464	579	734	892	964
231	231	233	236	238	240	241	242	243	243	245	320	554	1151	1153
255	233	236	239	249	336	411	450	463	473	486	518	705	919	1151

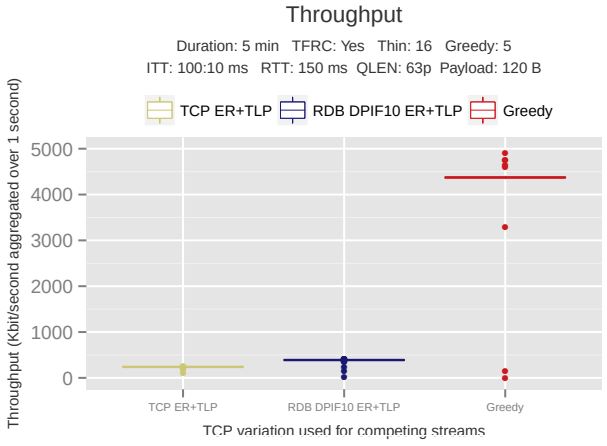
(c)

Figure A.3.97



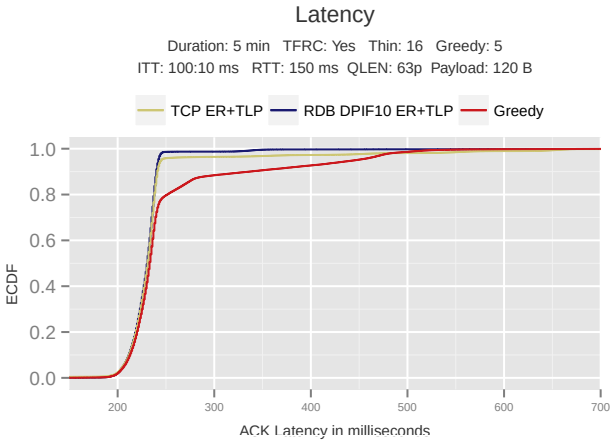
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	98 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF10 ER+TLP	rdbsender	16	99 / 100 / 101	239/240/241	12/1/7/9/10/108
Greedy	zsender	5	13 / 13 / 14	1447/1447/1447	17/4/14/16/20/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.7 K	454	469	0	5.8 M	5.8 M	56.4 K	56.4 K	0.93 / 0.93	1633
47.6 K	510	70	47.5 K	11.4 M	5.7 M	5.7 M	6.8 K	1.07 / 1.07	127
110.0 K	1173	1175	0	159.1 M	157.4 M	1.7 M	1.7 M	1.07 / 1.07	13694

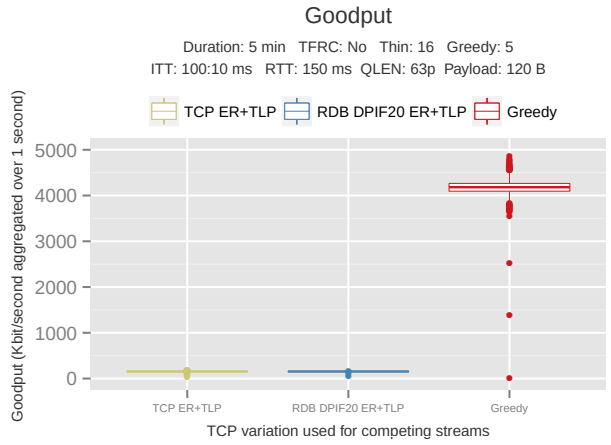
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
239	231	234	236	238	241	243	245	254	363	475	622	749	935	1048
231	231	233	235	238	240	241	242	243	244	245	331	737	1769	3884
254	233	236	239	251	336	407	446	461	471	483	511	694	927	1153

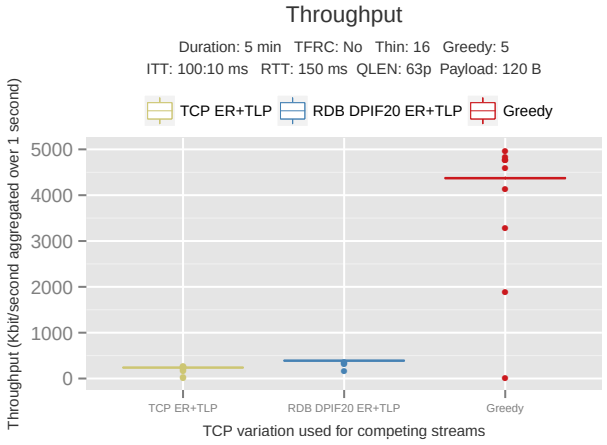
(c)

Figure A.3.98



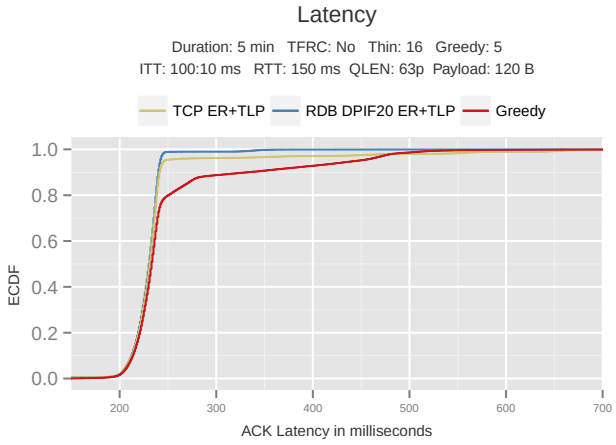
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	98 / 99 / 99	120/120/120	7/1/7/7/7/10
RDB DPIF20 ER+TLP	rdbsender	16	99 / 100 / 100	240/240/240	7/1/7/7/7/10
Greedy	zsender	5	13 / 13 / 14	1446/1447/1447	17/4/14/17/20/51

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.7 K	490	511	0	5.9 M	5.8 M	61.0 K	61.0 K	1.00 / 1.00	1717
47.7 K	453	21	47.7 K	11.4 M	5.7 M	5.7 M	2.5 K	0.95 / 0.95	55
110.1 K	1112	1114	0	159.3 M	157.6 M	1.6 M	1.6 M	1.01 / 1.01	13392

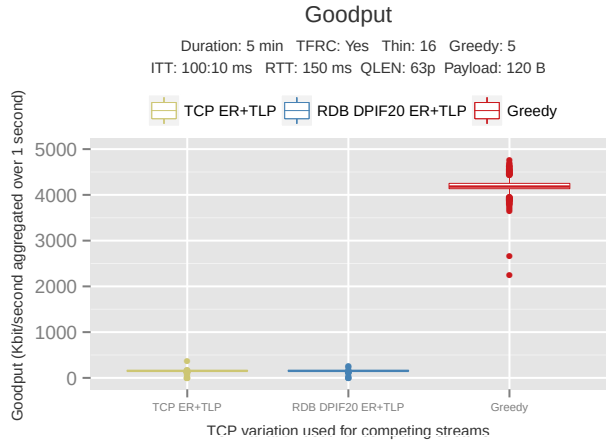
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
240	231	234	236	238	241	243	246	270	377	509	636	775	944	978
230	230	233	235	238	240	241	242	243	243	245	316	429	756	791
254	233	236	239	251	332	404	446	461	471	482	511	696	836	1153

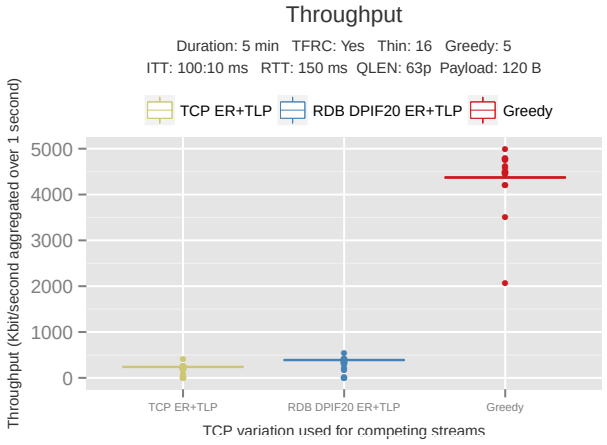
(c)

Figure A.3.99



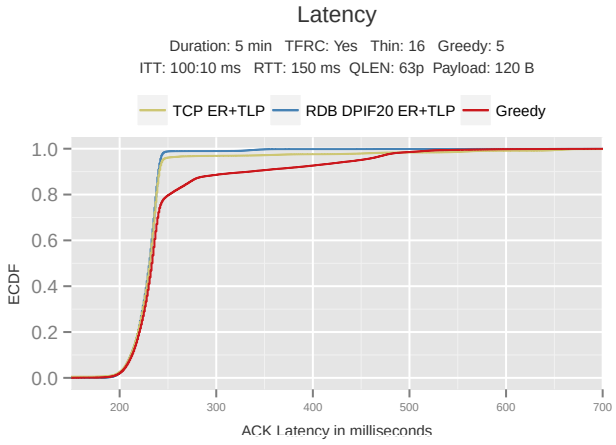
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	16	100 / 101 / 103	120/121/123	7/1/7/7/7/10
RDB DPIF20 ER+TLP	rdbsender	16	100 / 101 / 103	240/240/243	11/1/8/9/12/68
Greedy	zsender	5	12 / 13 / 14	1446/1447/1447	17/5/14/16/20/4

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.1 K	399	416	0	5.8 M	5.8 M	47.0 K	47.0 K	0.79 / 0.83	1421
47.0 K	430	50	47.0 K	11.3 M	5.7 M	5.6 M	3.4 K	0.89 / 0.91	67
110.2 K	1087	1087	0	159.5 M	157.9 M	1.6 M	1.6 M	0.99 / 0.99	13485

(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
238	231	234	236	238	241	243	245	249	331	457	572	752	3268	7757
231	231	233	236	238	240	241	242	243	243	245	314	569	3262	7721
254	233	236	240	251	334	407	448	462	472	484	515	664	743	999

(c)

Figure A.3.100

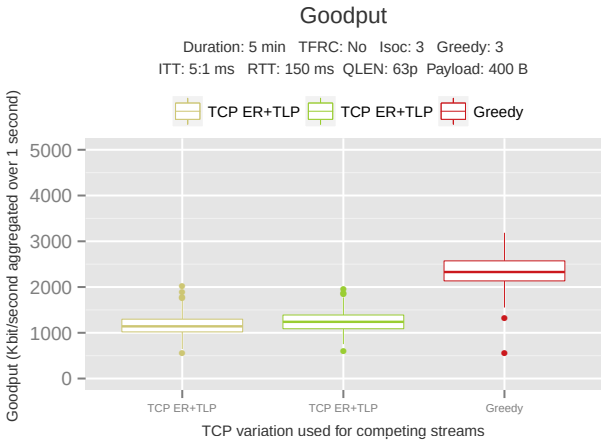
A.4 Fairness experiments

Table A.4 gives the complete test setup for the fairness experiments.

Table A.4

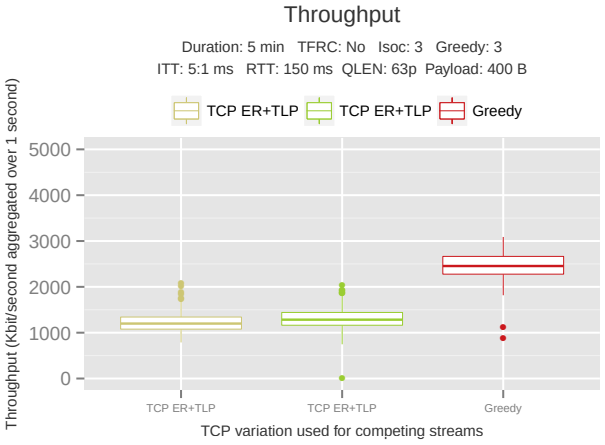
	Type	Streams	Cong	TFRC	RTT	ITT	Payload
Figure A.4.1	TCP ER+TLP	3	Rdb	No	150	5:1	400
Figure A.4.2	RDB DPIF10 ER+TLP	3	Rdb	No	150	5:1	400
Figure A.4.3	RDB DPIF10 ER+TLP	3	Rdb	Yes	150	5:1	400
Figure A.4.4	TCP ER+TLP	3	Rdb	No	150	15:1	400
Figure A.4.5	RDB DPIF10 ER+TLP	3	Rdb	No	150	15:1	400
Figure A.4.6	RDB DPIF10 ER+TLP	3	Rdb	Yes	150	15:1	400
Figure A.4.7	TCP ER+TLP	3	Rdb	No	150	30:3	400
Figure A.4.8	RDB DPIF10 ER+TLP	3	Rdb	No	150	30:3	400
Figure A.4.9	RDB DPIF10 ER+TLP	3	Rdb	Yes	150	30:3	400
Figure A.4.10	TCP ER+TLP	5	Rdb	No	150	5:1	400
Figure A.4.11	RDB DPIF10 ER+TLP	5	Rdb	No	150	5:1	400
Figure A.4.12	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	5:1	400
Figure A.4.13	TCP ER+TLP	5	Rdb	No	150	15:1	400
Figure A.4.14	RDB DPIF10 ER+TLP	5	Rdb	No	150	15:1	400
Figure A.4.15	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	15:1	400
Figure A.4.16	TCP ER+TLP	5	Rdb	No	150	30:3	400
Figure A.4.17	RDB DPIF10 ER+TLP	5	Rdb	No	150	30:3	400
Figure A.4.18	RDB DPIF10 ER+TLP	5	Rdb	Yes	150	30:3	400
Figure A.4.19	TCP ER+TLP	7	Rdb	No	150	5:1	400
Figure A.4.20	RDB DPIF10 ER+TLP	7	Rdb	No	150	5:1	400
Figure A.4.21	RDB DPIF10 ER+TLP	7	Rdb	Yes	150	5:1	400
Figure A.4.22	TCP ER+TLP	7	Rdb	No	150	15:1	400
Figure A.4.23	RDB DPIF10 ER+TLP	7	Rdb	No	150	15:1	400
Figure A.4.24	RDB DPIF10 ER+TLP	7	Rdb	Yes	150	15:1	400
Figure A.4.25	TCP ER+TLP	7	Rdb	No	150	30:3	400
Figure A.4.26	RDB DPIF10 ER+TLP	7	Rdb	No	150	30:3	400
Figure A.4.27	RDB DPIF10 ER+TLP	7	Rdb	Yes	150	30:3	400
Figure A.4.28	TCP ER+TLP	10	Rdb	No	150	5:1	400
Figure A.4.29	RDB DPIF10 ER+TLP	10	Rdb	No	150	5:1	400
Figure A.4.30	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	5:1	400
Figure A.4.31	TCP ER+TLP	10	Rdb	No	150	15:1	400
Figure A.4.32	RDB DPIF10 ER+TLP	10	Rdb	No	150	15:1	400
Figure A.4.33	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	15:1	400
Figure A.4.34	TCP ER+TLP	10	Rdb	No	150	30:3	400
Figure A.4.35	RDB DPIF10 ER+TLP	10	Rdb	No	150	30:3	400
Figure A.4.36	RDB DPIF10 ER+TLP	10	Rdb	Yes	150	30:3	400
Figure A.4.37	TCP ER+TLP	13	Rdb	No	150	5:1	400
Figure A.4.38	RDB DPIF10 ER+TLP	13	Rdb	No	150	5:1	400
Figure A.4.39	RDB DPIF10 ER+TLP	13	Rdb	Yes	150	5:1	400
Figure A.4.40	TCP ER+TLP	13	Rdb	No	150	15:1	400
Figure A.4.41	RDB DPIF10 ER+TLP	13	Rdb	No	150	15:1	400
Figure A.4.42	RDB DPIF10 ER+TLP	13	Rdb	Yes	150	15:1	400
Figure A.4.43	TCP ER+TLP	13	Rdb	No	150	30:3	400
Figure A.4.44	RDB DPIF10 ER+TLP	13	Rdb	No	150	30:3	400
Figure A.4.45	RDB DPIF10 ER+TLP	13	Rdb	Yes	150	30:3	400

Table A.4: Test setup for experiment 4



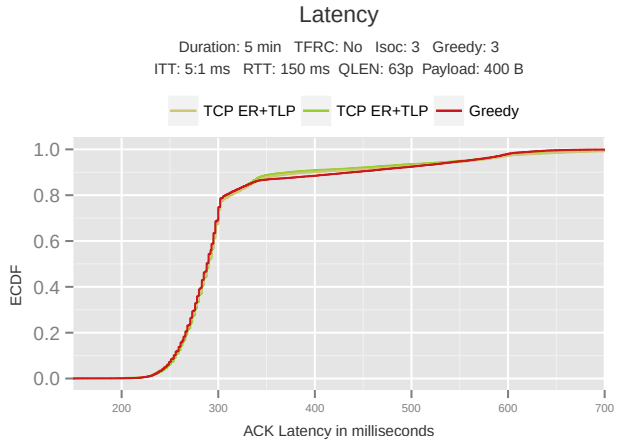
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	26 / 28 / 30	1394/1424/1444	11/1/8/10/13/34
TCP ER+TLP	rdbsender	3	26 / 27 / 27	1404/1420/1433	12/1/8/11/14/42
Greedy	zsender	3	14 / 14 / 15	1446/1447/1448	20/7/16/19/23/38

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
31.6 K	400	400	0	44.9 M	44.4 M	546 K	546 K	1.21 / 1.27	3410
33.3 K	378	378	0	47.3 M	46.8 M	511 K	511 K	1.08 / 1.14	3429
61.9 K	614	614	0	89.6 M	88.7 M	888 K	888 K	0.99 / 0.99	8323

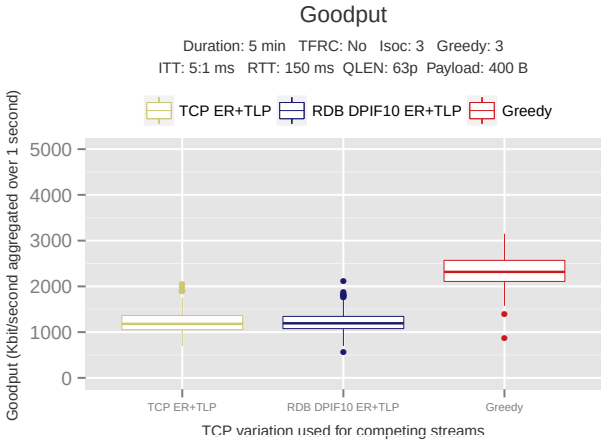
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
314	290	295	300	313	397	504	562	582	598	629	680	808	932	999
312	290	295	300	311	372	482	550	574	593	619	667	788	999	1120
313	288	295	300	307	438	513	557	574	588	600	624	691	786	999

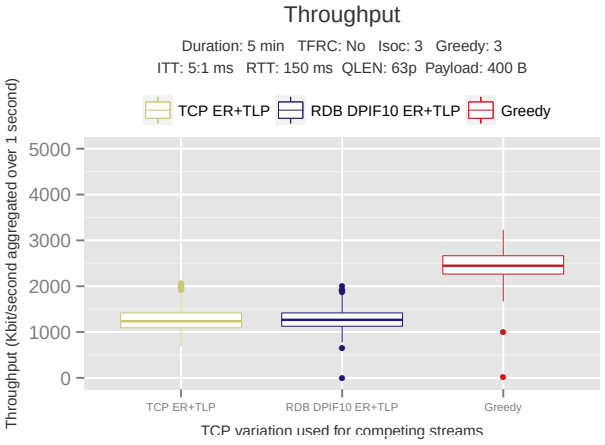
(c)

Figure A.4.1



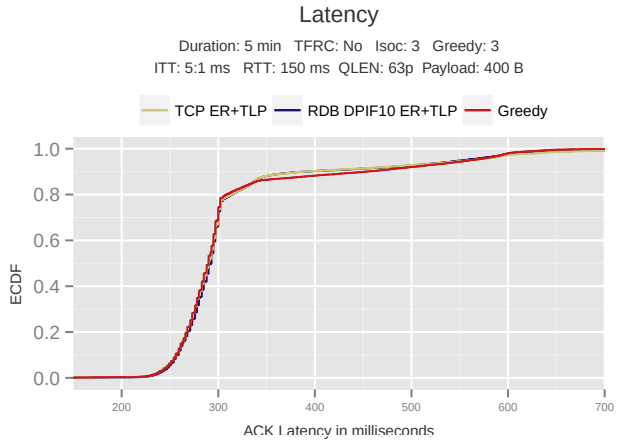
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	26 / 28 / 30	1387/1421/1444	11/1/8/11/14/36
RDB DPIF10 ER+TLP	rdbsender	3	26 / 28 / 30	1442/1442/1443	11/1/8/11/14/43
Greedy	zsender	3	14 / 14 / 14	1447/1447/1447	20/6/16/19/24/48

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
32.6 K	394	394	0	46.3 M	45.8 M	548 K	548 K	1.18 / 1.21	3412
32.4 K	393	393	62	46.7 M	46.1 M	583 K	537 K	1.15 / 1.21	3427
61.5 K	610	610	0	89.0 M	88.1 M	881 K	881 K	0.99 / 0.99	8533

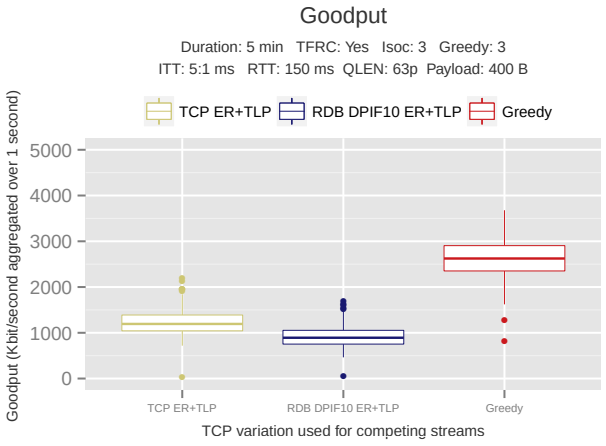
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
314	290	295	300	313	389	506	561	583	598	629	681	815	1036	1048
314	293	297	300	314	390	509	554	574	593	612	670	785	974	1031
315	290	295	300	309	453	523	561	578	591	600	627	709	1069	1362

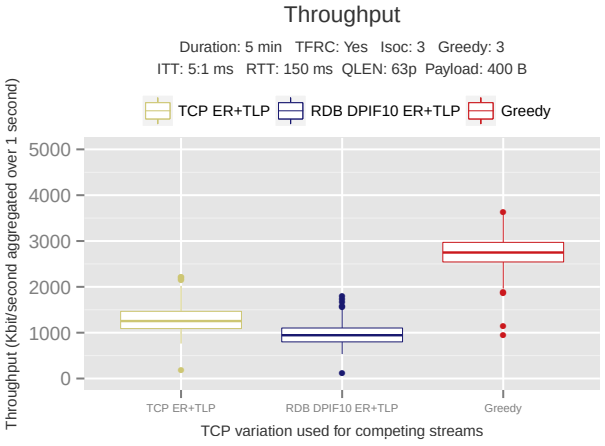
(c)

Figure A.4.2



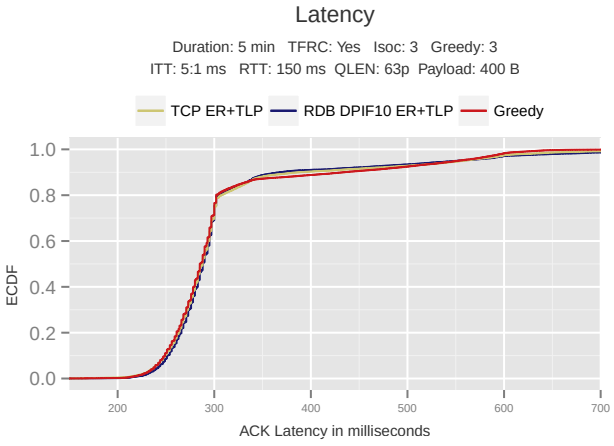
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	26 / 27 / 27	1405/1416/1426	12/1/8/11/14/34
RDB DPIF10 ER+TLP	rdbsender	3	36 / 37 / 37	1418/1432/1443	9/1/6/8/12/33
Greedy	zsender	3	12 / 13 / 13	1446/1446/1447	22/1/17/22/27/46

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
33.3 K	412	412	0	47.1 M	46.5 M	541 K	541 K	1.15 / 1.24	3487
24.2 K	313	304	67	34.7 M	34.3 M	453 K	403 K	1.20 / 1.29	2249
69.1 K	598	598	0	99.9 M	99.0 M	865 K	865 K	0.87 / 0.87	9127

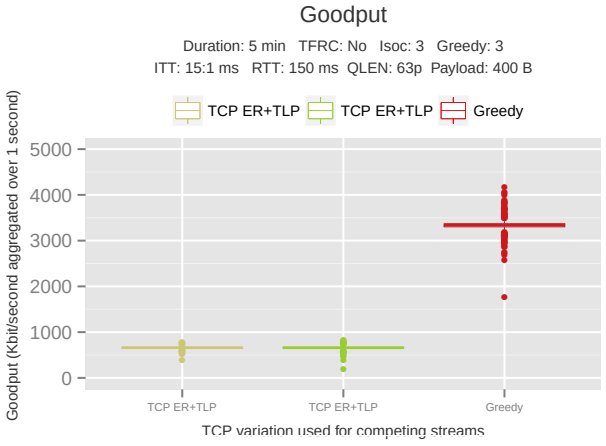
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
310	288	294	300	307	388	505	557	576	595	623	680	832	999	1120
313	288	295	300	305	368	484	552	581	600	651	734	1185	1998	2228
310	285	293	297	302	434	511	552	571	586	598	620	707	3893	6012

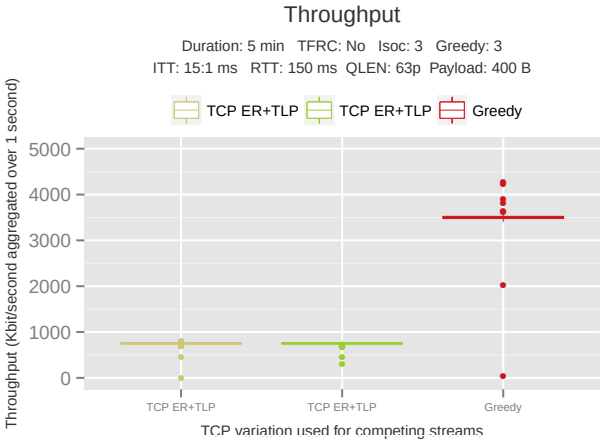
(c)

Figure A.4.3



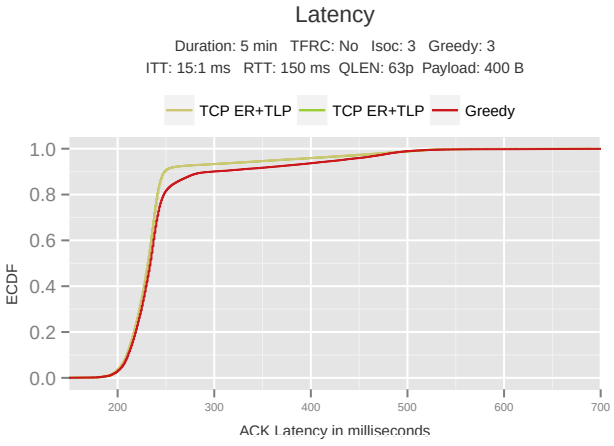
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	16 / 17 / 17	457/464/474	17/3/13/18/21/22
TCP ER+TLP	rdbsender	3	16 / 17 / 17	457/462/467	17/3/13/18/21/22
Greedy	zsender	3	10 / 10 / 10	1447/1447/1448	23/7/19/22/26/61

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
53.6 K	302	302	0	24.9 M	24.7 M	123 K	123 K	0.49 / 0.56	3980
53.2 K	301	301	0	24.5 M	24.4 M	128 K	128 K	0.52 / 0.57	3952
87.7 K	660	660	0	127 M	126 M	956 K	956 K	0.75 / 0.75	9454

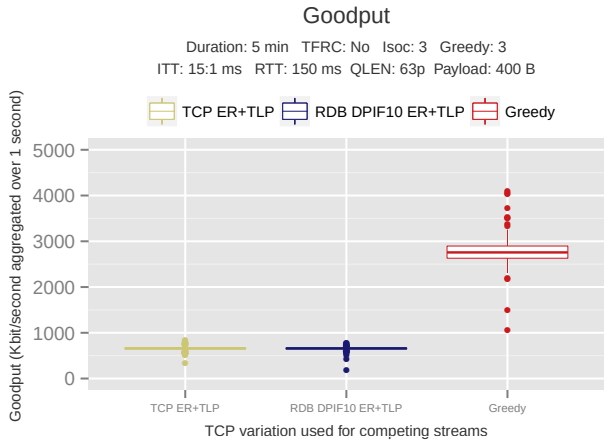
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
241	231	235	238	241	249	287	363	402	436	471	503	554	613	650
242	232	235	238	241	249	288	365	404	442	475	507	625	787	999
251	234	237	241	248	298	384	431	451	468	482	503	676	732	999

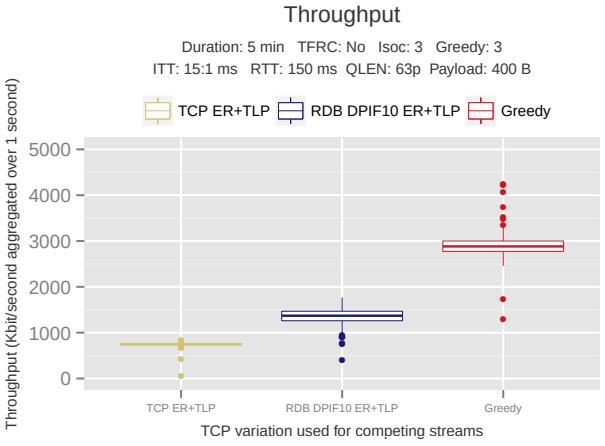
(c)

Figure A.4.4



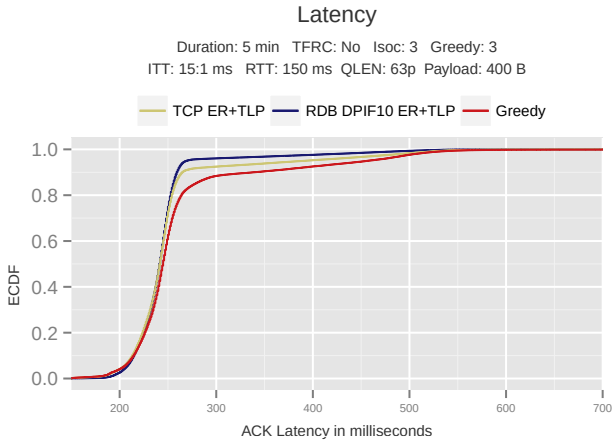
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	17 / 17 / 18	481/488/496	16/1/13/17/21/23
RDB DPIF10 ER+TLP	rdbsender	3	17 / 17 / 18	898/940/1023	16/2/13/17/21/23
Greedy	zsender	3	12 / 12 / 13	1446/1447/1448	20/1/16/19/23/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
51.1 K	342	342	0	24.9 M	24.8 M	144 K	144 K	0.58 / 0.67	4202
50.4 K	330	160	30.9 K	47.4 M	24.4 M	23.0 M	80.9 K	0.58 / 0.66	2266
73.1 K	723	723	0	106 M	105 M	1.0 M	1.0 M	0.99 / 0.99	8729

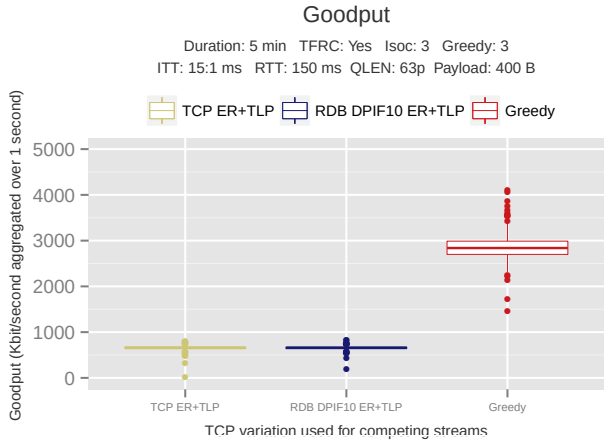
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
252	242	245	249	254	265	319	392	427	461	491	520	584	671	713
246	241	245	249	253	259	263	269	296	361	423	479	545	616	999
264	245	249	255	264	339	411	457	476	491	506	529	729	1736	1749

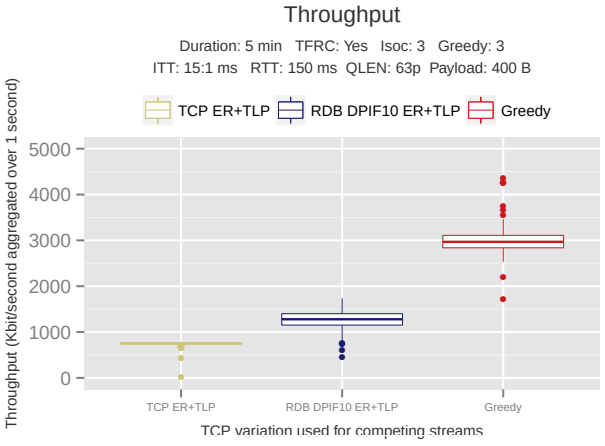
(c)

Figure A.4.5



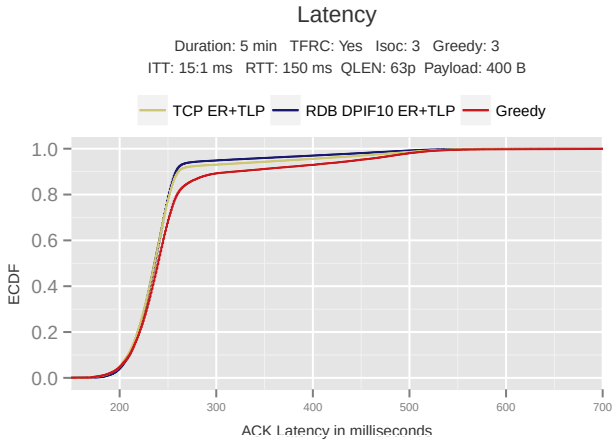
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	17 / 17 / 17	468/471/476	17/4/13/17/21/23
RDB DPIF10 ER+TLP	rdbsender	3	18 / 18 / 19	894/911/928	18/1/12/19/22/38
Greedy	zsender	3	10 / 12 / 13	1447/1447/1448	20/7/16/19/24/47

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
52.8 K	335	335	0	24.9 M	24.7 M	139 K	139 K	0.56 / 0.63	4079
48.6 K	347	212	26.4 K	44.2 M	24.4 M	19.8 M	104 K	0.60 / 0.71	2905
75.4 K	759	759	0	109 M	108 M	1.1 M	1.1 M	1.01 / 1.01	8545

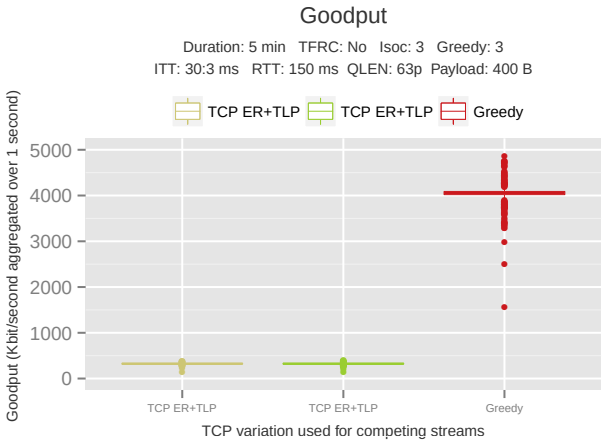
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
247	236	241	246	252	261	298	377	416	450	481	514	608	811	859
244	236	241	246	251	258	264	305	352	400	446	491	574	699	999
258	240	245	251	260	318	401	446	466	483	499	522	722	959	999

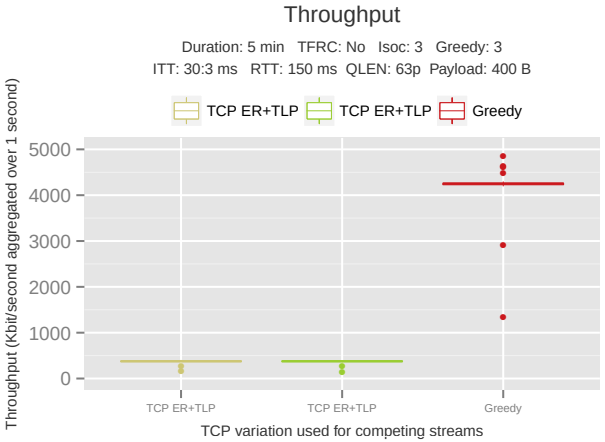
(c)

Figure A.4.6



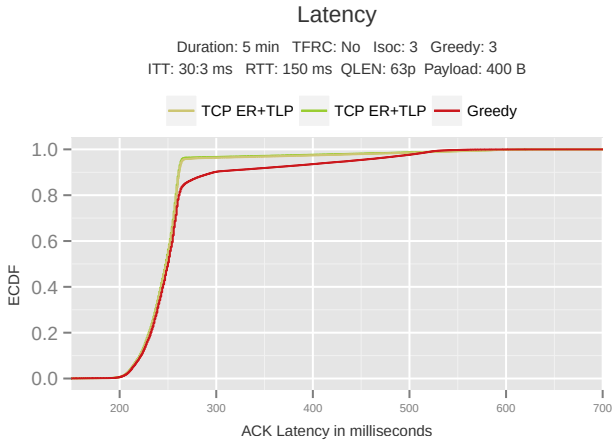
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	30 / 30 / 30	405/406/408	13/4/12/13/14/14
TCP ER+TLP	rdbsender	3	30 / 30 / 30	404/406/407	13/4/13/13/14/14
Greedy	zsender	3	8 / 8 / 9	1447/1447/1447	29/10/24/29/33/56

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.1 K	125	125	0	12.2 M	12.2 M	50.0 K	50.0 K	0.41 / 0.41	1145
29.8 K	110	110	0	12.1 M	12.1 M	44.0 K	44.0 K	0.36 / 0.37	1020
106 K	561	561	0	154 M	153 M	812 K	812 K	0.53 / 0.53	10931

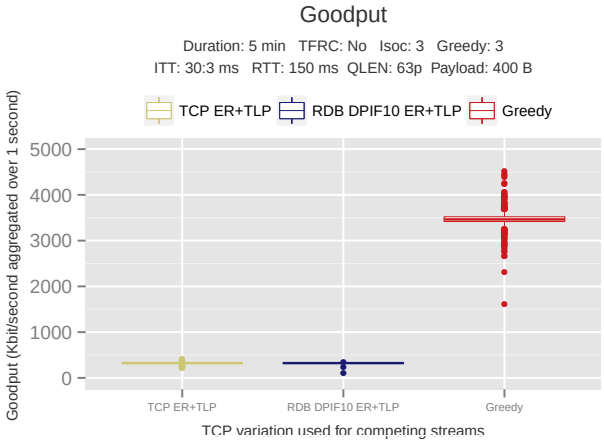
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
252	248	252	256	259	261	263	265	276	365	455	537	614	811	895
251	248	252	256	258	261	262	264	265	333	434	526	612	659	999
264	250	255	258	262	298	385	440	465	486	506	522	568	709	999

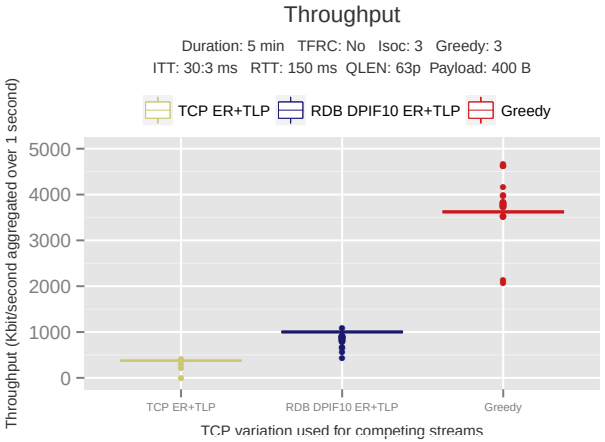
(c)

Figure A.4.7



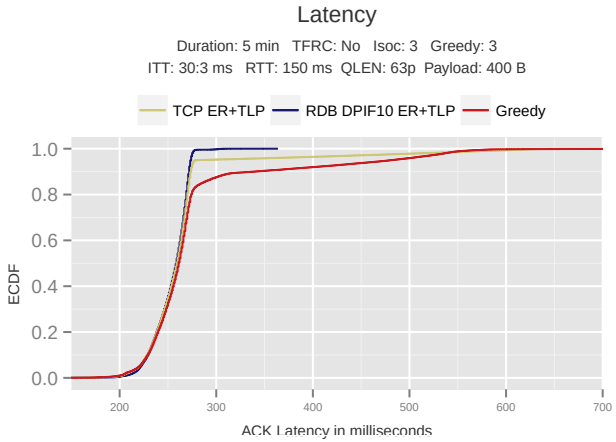
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	30 / 30 / 30	406/408/409	13/4/12/14/14/14
RDB DPIF10 ER+TLP	rdbsender	3	30 / 30 / 30	1190/1191/1192	13/3/12/14/14/14
Greedy	zsender	3	9 / 10 / 10	1446/1447/1447	26/10/22/26/29/46

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
30.0 K	159	159	0	12.2 M	12.2 M	63.6 K	63.6 K	0.52 / 0.53	1421
29.4 K	126	0	29.0 K	35.0 M	12.0 M	23.0 M	0	0.43 / 0.43	
91.3 K	594	594	0	132 M	131 M	860 K	860 K	0.65 / 0.65	10196

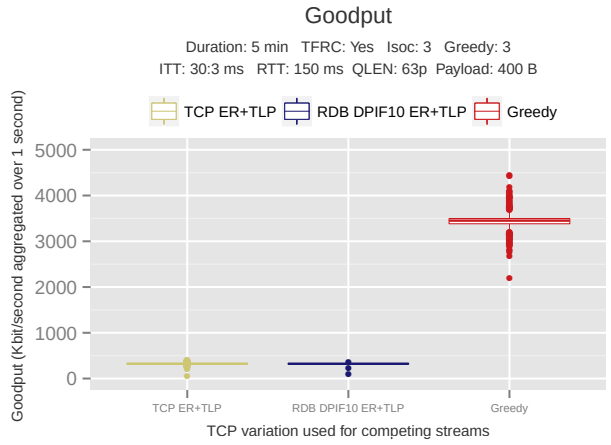
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
265	259	264	267	270	274	275	280	363	442	512	575	643	930	1009
254	259	263	266	269	272	273	274	274	275	275	277	305	324	348
278	261	266	270	274	339	430	481	502	521	537	554	722	807	999

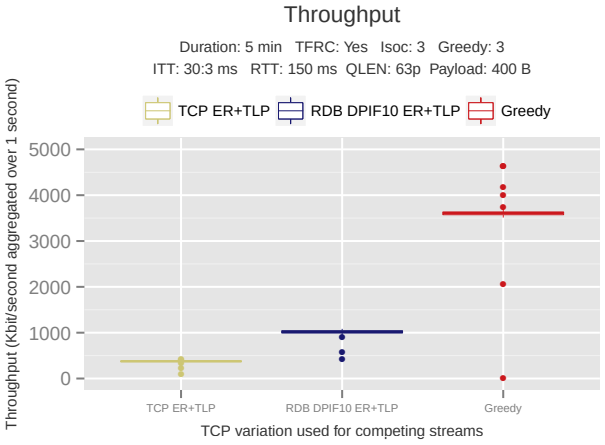
(c)

Figure A.4.8



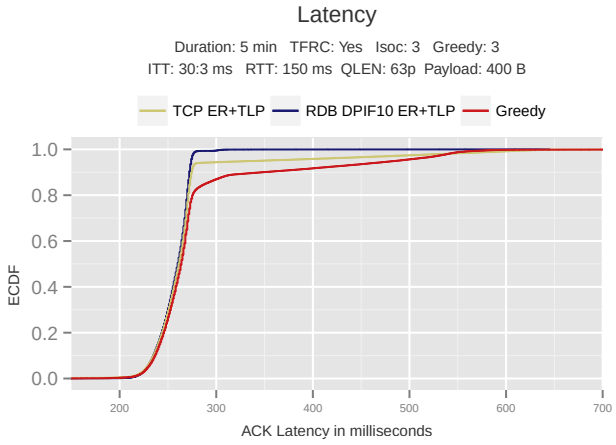
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	3	30 / 30 / 30	409/412/415	12/1/11/14/14/14
RDB DPIF10 ER+TLP	rdbsender	3	30 / 30 / 30	1199/1199/1200	16/3/14/16/18/22
Greedy	zsender	3	9 / 10 / 10	1447/1447/1448	26/10/22/26/30/57

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
29.8 K	180	180	0	12.3 M	12.2 M	72.0 K	72.0 K	0.59 / 0.60	1642
30.0 K	186	2	30.0 K	36.0 M	12.0 M	24.0 M	800	0.62 / 0.62	19
90.4 K	606	606	0	131 M	130 M	875 K	875 K	0.67 / 0.67	10494

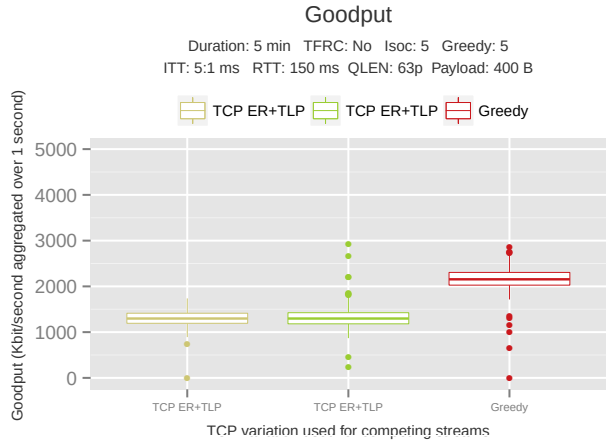
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
270	261	266	269	271	275	277	343	412	476	536	591	669	1015	1104
257	260	264	268	270	272	273	274	275	275	276	279	312	558	609
282	264	268	271	275	348	436	487	508	527	540	557	698	813	999

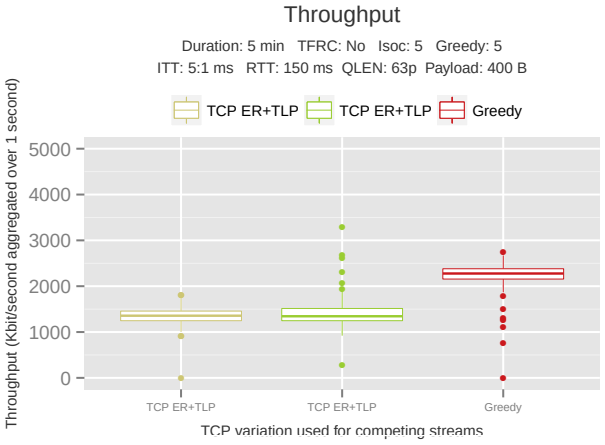
(c)

Figure A.4.9



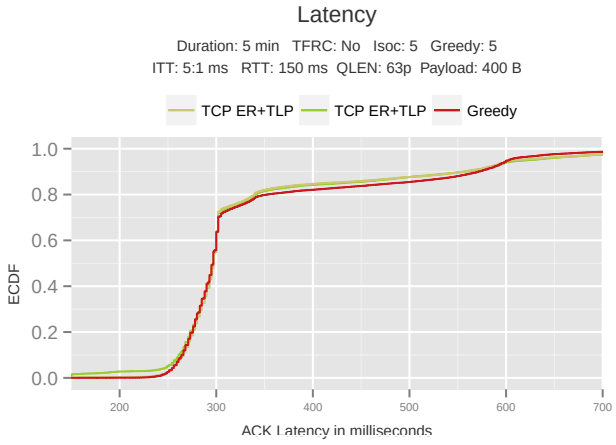
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	42 / 43 / 44	1442/1445/1446	7/1/5/7/9/26
TCP ER+TLP	rdbsender	5	39 / 41 / 42	1410/1411/1412	9/1/5/7/9/52
Greedy	zsender	5	25 / 26 / 27	1446/1447/1447	11/1/9/11/13/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
34.9 K	883	859	0	50.4 M	49.2 M	1.2 M	1.2 M	2.51 / 2.53	5399
36.5 K	1030	1000	0	51.5 M	50.1 M	1.3 M	1.3 M	2.67 / 2.82	5599
57.6 K	1308	1310	0	83.3 M	81.4 M	1.9 M	1.9 M	2.27 / 2.27	10740

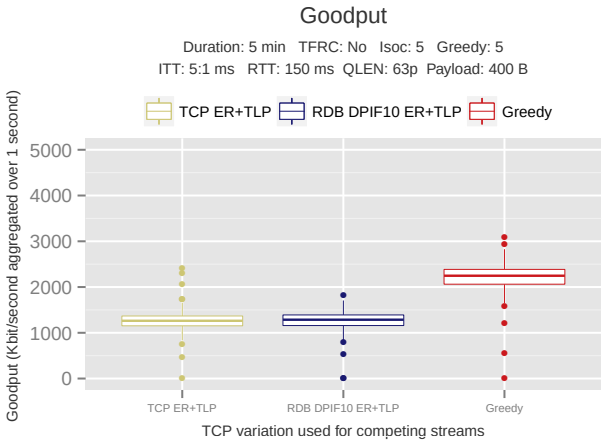
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
339	297	300	302	339	557	591	608	641	679	724	779	903	1151	1172
337	297	300	302	341	557	593	624	649	682	726	775	913	1138	1187
343	297	300	302	353	571	593	603	612	634	668	724	884	2634	4141

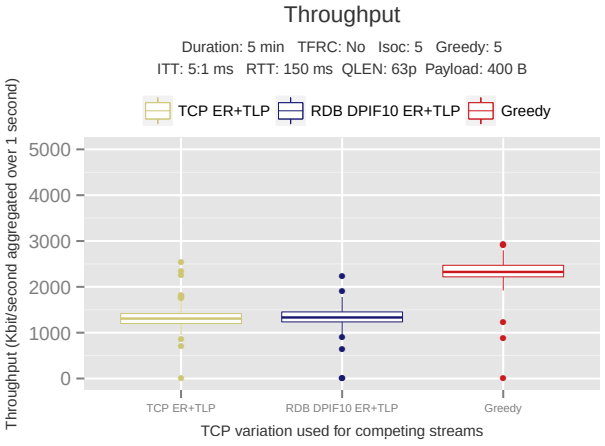
(c)

Figure A.4.10



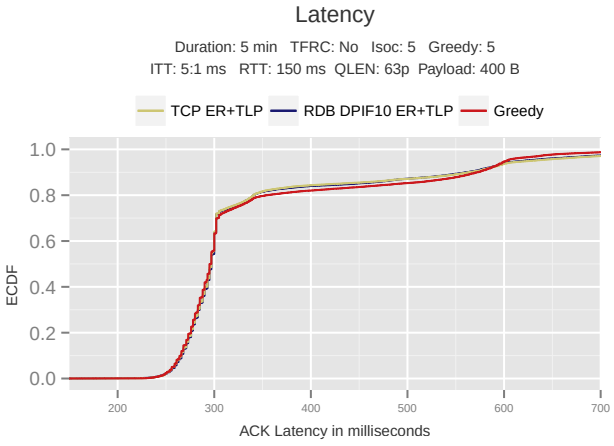
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	43 / 44 / 45	1446/1446/1446	7/1/5/7/8/18
RDB DPIF10 ER+TLP	rdbsender	5	42 / 44 / 45	1440/1442/1443	7/1/5/7/8/35
Greedy	zsender	5	24 / 25 / 26	1445/1446/1447	12/5/9/11/13/32

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
34.0 K	895	897	0	49.2 M	47.9 M	1.3 M	1.3 M	2.63 / 2.63	5204
34.0 K	917	905	113	49.0 M	47.7 M	1.3 M	1.3 M	2.60 / 2.70	5297
59.6 K	1317	1319	0	86.3 M	84.4 M	1.9 M	1.9 M	2.20 / 2.21	11326

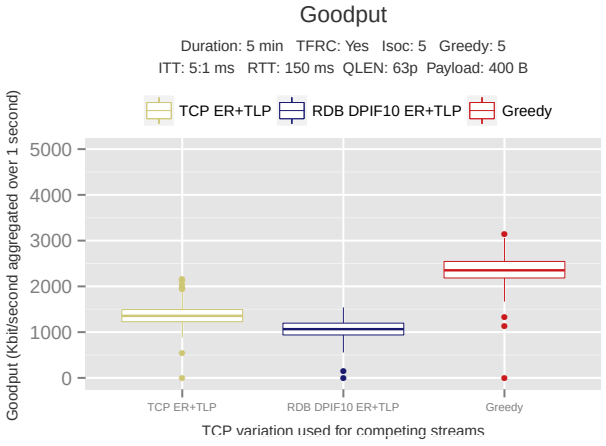
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
342	297	300	302	341	564	595	627	658	695	743	787	999	4091	4170
341	297	300	302	341	561	593	620	649	686	734	779	892	1130	1423
342	296	300	304	355	572	593	603	612	634	658	721	847	1071	1150

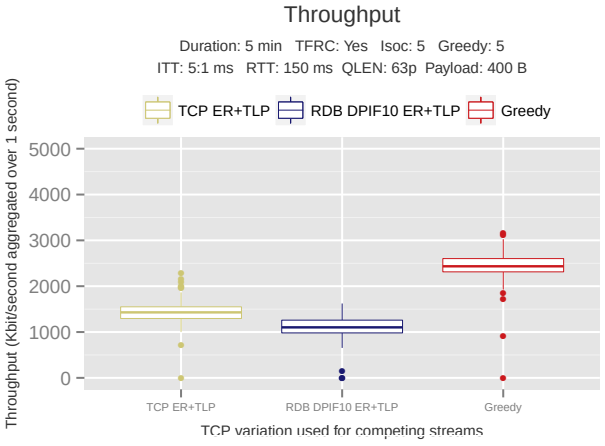
(c)

Figure A.4.11



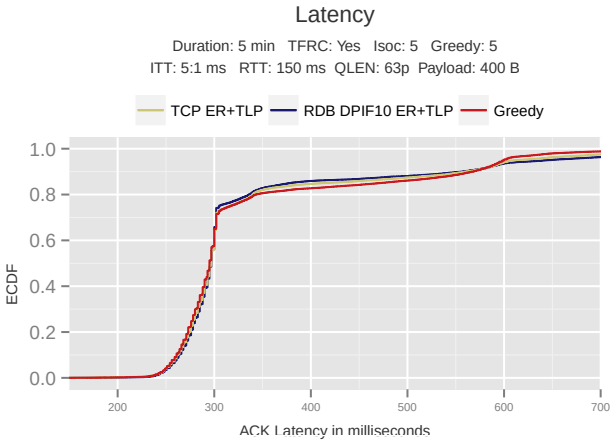
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	38 / 41 / 43	1444/1445/1446	7/1/6/7/9/25
RDB DPIF10 ER+TLP	rdbsender	5	47 / 53 / 56	1444/1445/1445	6/1/4/6/8/28
Greedy	zsender	5	23 / 24 / 25	1445/1446/1447	12/4/10/12/14/30

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
36.7 K	899	901	0	53.0 M	51.7 M	1.3 M	1.3 M	2.43 / 2.45	5451
28.3 K	727	719	83	40.9 M	39.9 M	1.1 M	1.0 M	2.53 / 2.57	3808
62.5 K	1300	1302	0	90.4 M	88.5 M	1.9 M	1.9 M	2.08 / 2.08	11344

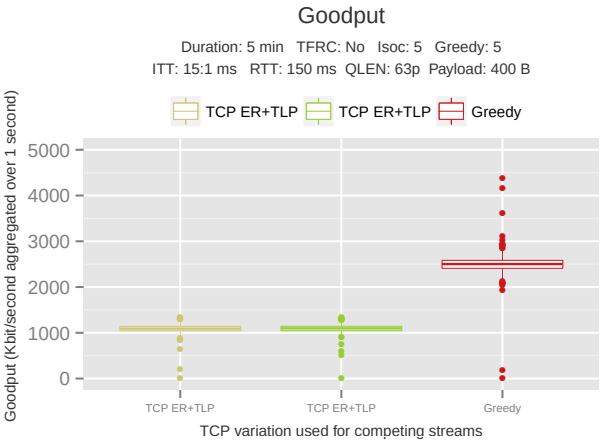
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
338	297	300	302	340	559	593	612	644	681	725	775	900	1101	1152
341	297	300	302	336	557	595	646	685	739	784	850	1341	1810	1922
338	295	300	302	343	568	590	600	605	627	652	714	847	1043	1153

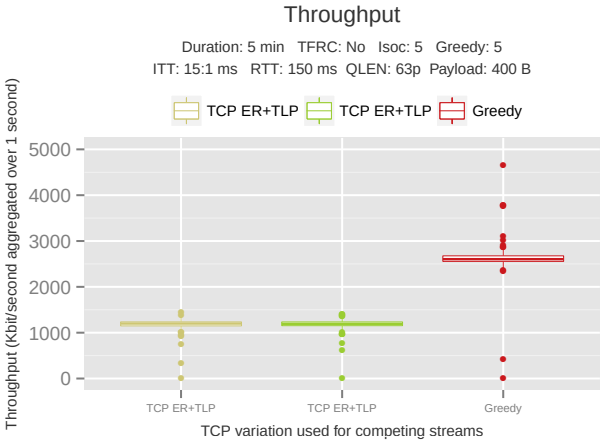
(c)

Figure A.4.12



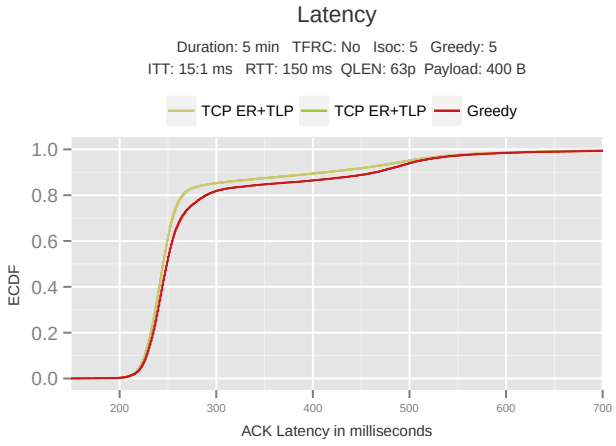
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	25 / 27 / 28	701/746/794	11/1/8/11/14/22
TCP ER+TLP	rdbsender	5	24 / 27 / 29	686/751/813	11/1/8/11/14/23
Greedy	zsender	5	21 / 22 / 23	1446/1447/1447	11/3/9/11/13/29

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
56.1 K	972	974	0	41.7 M	41.1 M	659 K	659 K	1.58 / 1.73	7985
55.2 K	992	994	0	41.3 M	40.6 M	692 K	692 K	1.68 / 1.80	7817
67.1 K	1509	1511	0	97.1 M	94.9 M	2.2 M	2.2 M	2.25 / 2.25	10885

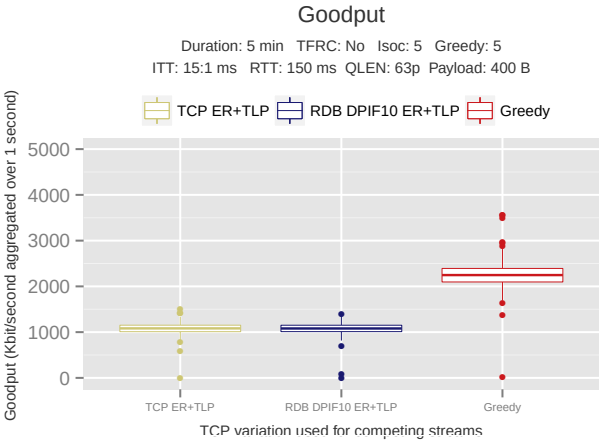
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
276	245	249	255	265	411	470	498	513	533	563	628	873	1152	1405
277	245	249	255	265	413	470	499	515	535	567	646	1048	1563	1641
287	249	254	264	289	465	492	510	524	542	576	652	868	1091	1469

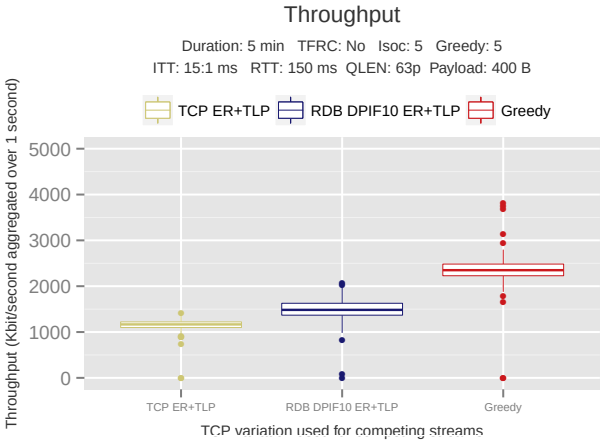
(c)

Figure A.4.13



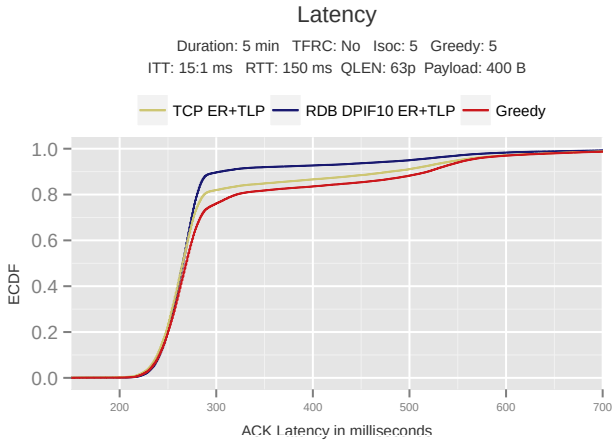
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	30 / 33 / 35	849/914/977	10/1/7/9/12/22
RDB DPIF10 ER+TLP	rdbsender	5	31 / 33 / 35	1173/1200/1213	10/1/7/9/12/23
Greedy	zsender	5	23 / 25 / 27	1446/1447/1448	11/1/9/10/13/30

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
45.6 K	978	980	0	41.6 M	40.8 M	772 K	772 K	1.86 / 2.14	6988
44.5 K	938	545	18.2 K	53.4 M	40.2 M	13.3 M	635 K	2.09 / 2.11	3487
60.4 K	1534	1536	0	87.4 M	85.2 M	2.2 M	2.2 M	2.54 / 2.54	10575

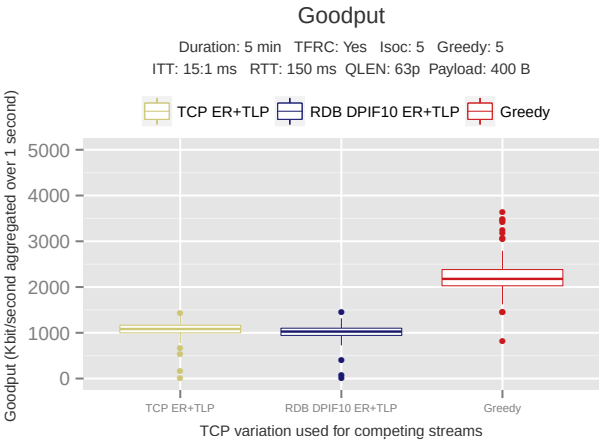
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
302	265	271	277	288	480	525	551	568	595	644	715	985	1524	1891
286	266	270	275	280	304	420	501	525	549	583	667	957	1286	1993
314	269	275	284	321	519	542	560	574	601	651	725	1006	2105	2115

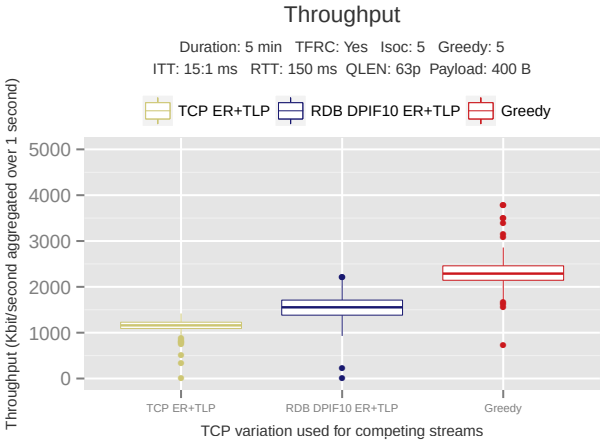
(c)

Figure A.4.14



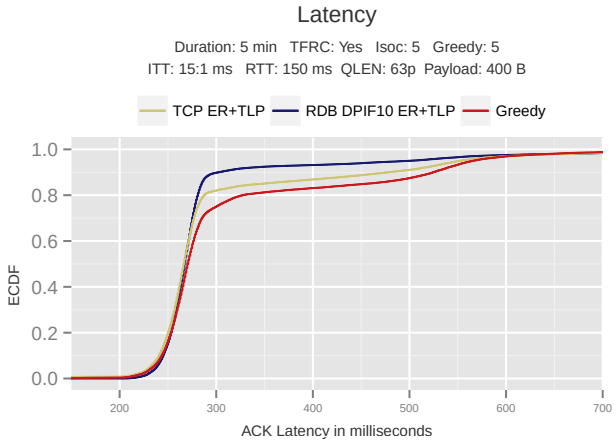
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	30 / 33 / 35	840/901/946	10/1/7/9/13/22
RDB DPIF10 ER+TLP	rdbsender	5	28 / 33 / 40	1194/1234/1286	12/1/7/12/15/50
Greedy	zsender	5	23 / 25 / 26	1446/1447/1447	11/3/9/10/13/44

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
45.8 K	980	982	0	41.2 M	40.4 M	799 K	799 K	1.94 / 2.14	6760
45.5 K	1119	584	23.5 K	56.1 M	38.1 M	17.9 M	734 K	2.48 / 2.46	3056
59.3 K	1581	1581	0	85.7 M	83.4 M	2.3 M	2.3 M	2.67 / 2.67	10662

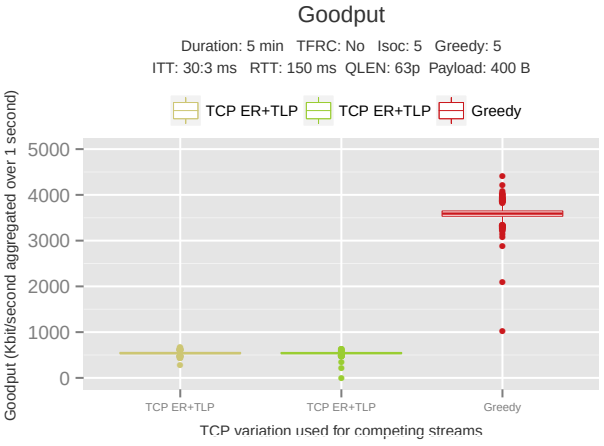
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
304	266	271	277	288	478	528	552	569	599	661	730	1145	3979	4043
290	267	271	275	281	302	388	499	531	565	653	753	1273	1744	2022
316	270	276	285	327	525	548	566	580	602	644	719	980	1073	1245

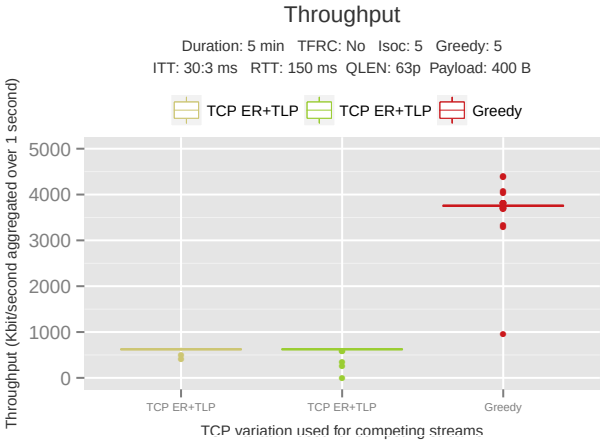
(c)

Figure A.4.15



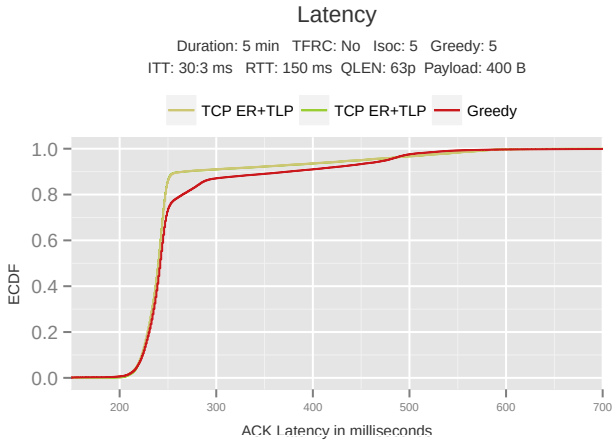
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	31 / 31 / 32	426/432/440	11/1/8/12/13/14
TCP ER+TLP	rdbsender	5	31 / 32 / 32	426/432/437	11/1/8/12/13/14
Greedy	zsender	5	15 / 16 / 16	1446/1447/1447	16/4/12/15/18/42

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
47.6 K	540	540	0	20.6 M	20.3 M	220 K	220 K	1.07 / 1.14	4329
46.8 K	534	536	0	20.2 M	20.0 M	220 K	220 K	1.08 / 1.14	4276
94.7 K	1164	1164	0	137 M	135 M	1.7 M	1.7 M	1.23 / 1.23	12427

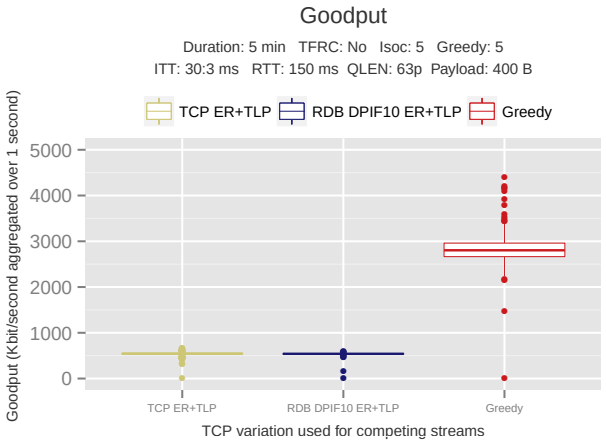
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
258	240	242	245	247	266	377	448	481	511	542	571	656	805	863
258	240	242	245	247	267	380	451	482	512	541	570	634	767	1264
267	242	245	248	265	375	445	476	485	493	509	539	718	816	999

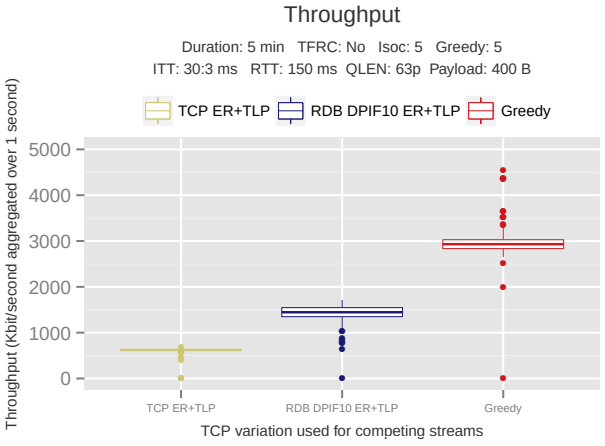
(c)

Figure A.4.16



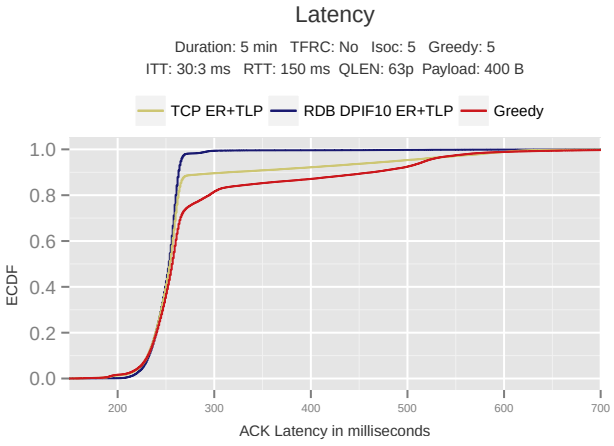
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	32 / 32 / 32	434/439/444	11/1/8/11/13/14
RDB DPIF10 ER+TLP	rdbsender	5	33 / 34 / 34	1157/1160/1166	10/1/8/10/13/14
Greedy	zsender	5	18 / 20 / 22	1446/1447/1447	13/4/10/12/15/37

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
46.8 K	589	591	0	20.6 M	20.3 M	242 K	242 K	1.18 / 1.26	4672
43.8 K	597	33	40.1 K	50.9 M	20.0 M	30.8 M	24.8 K	1.38 / 1.36	181
75.4 K	1439	1441	0	109 M	107 M	2.1 M	2.1 M	1.91 / 1.91	11714

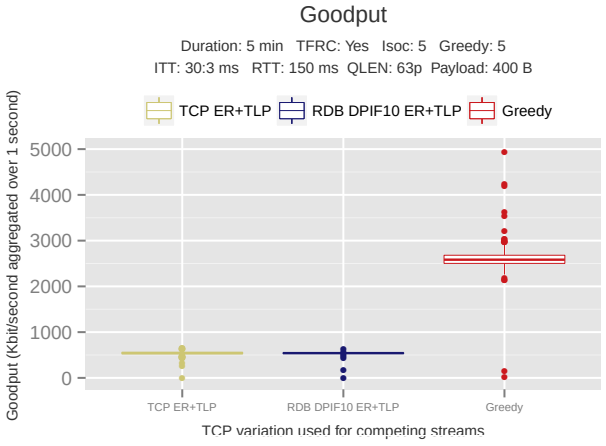
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
273	254	257	260	263	315	428	491	518	548	577	604	698	965	1150
252	253	256	258	260	263	264	266	266	268	271	292	615	843	1151
290	257	260	265	295	463	505	519	528	542	563	608	800	1070	1152

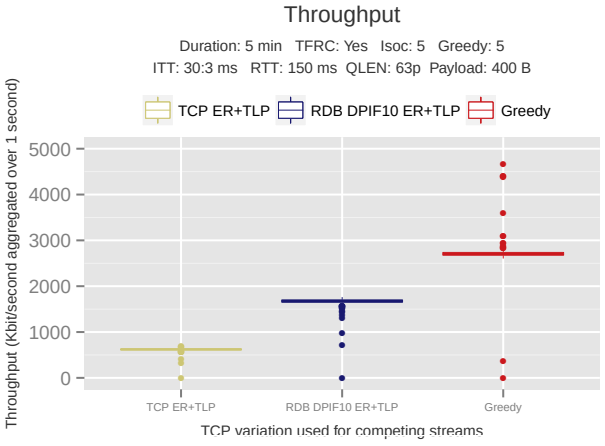
(c)

Figure A.4.17



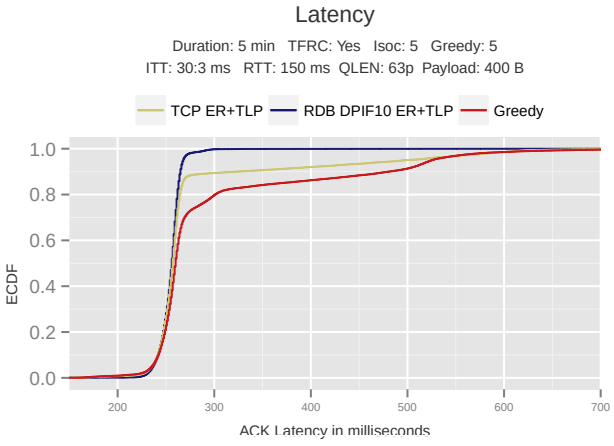
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	5	32 / 33 / 34	439/450/465	11/1/8/11/13/14
RDB DPIF10 ER+TLP	rdbsender	5	30 / 30 / 30	1194/1197/1199	13/1/12/13/14/24
Greedy	zsender	5	20 / 21 / 23	1446/1447/1447	12/4/10/12/14/41

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
45.7 K	581	583	0	20.6 M	20.3 M	252 K	252 K	1.23 / 1.27	4621
49.6 K	690	9	49.4 K	59.4 M	20.0 M	39.4 M	4.6 K	1.39 / 1.39	42
69.4 K	1343	1345	0	100 M	98.5 M	1.9 M	1.9 M	1.94 / 1.93	11607

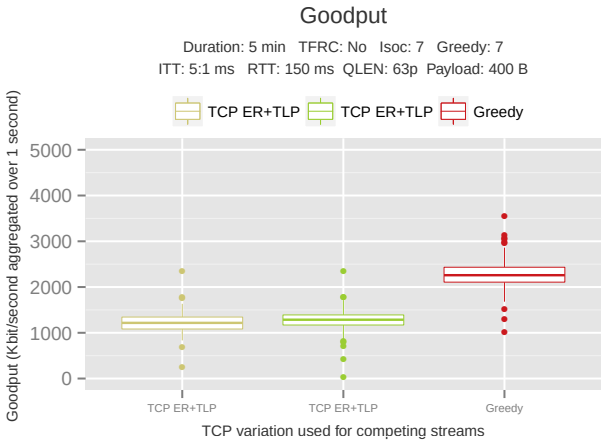
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
279	257	259	261	264	323	434	498	525	555	584	614	779	1076	1238
256	256	258	260	262	265	266	268	269	271	275	289	353	1065	1621
298	260	263	269	301	481	513	525	536	551	575	629	822	1014	1152

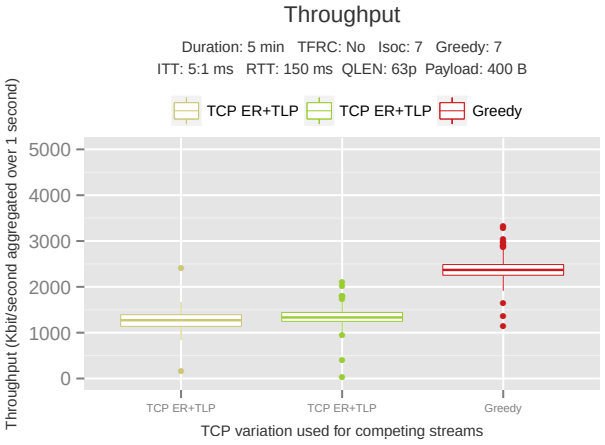
(c)

Figure A.4.18



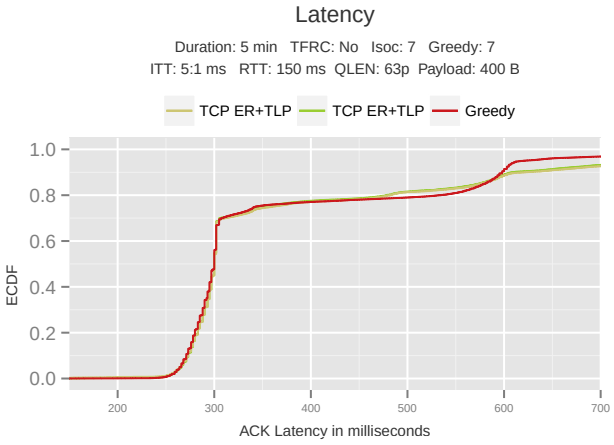
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	61 / 63 / 65	1444/1445/1445	5/1/4/5/6/17
TCP ER+TLP	rdbsender	7	58 / 60 / 63	1441/1443/1445	5/1/4/5/7/24
Greedy	zsender	7	31 / 34 / 37	1444/1446/1447	9/3/7/8/10/25

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
33.4 K	1481	1481	0	48.2 M	46.1 M	2.1 M	2.1 M	4.43 / 4.43	7143
35.0 K	1498	1498	0	50.5 M	48.4 M	2.1 M	2.1 M	4.23 / 4.28	7222
61.4 K	2106	2106	0	88.8 M	85.8 M	3.0 M	3.0 M	3.42 / 3.43	13775

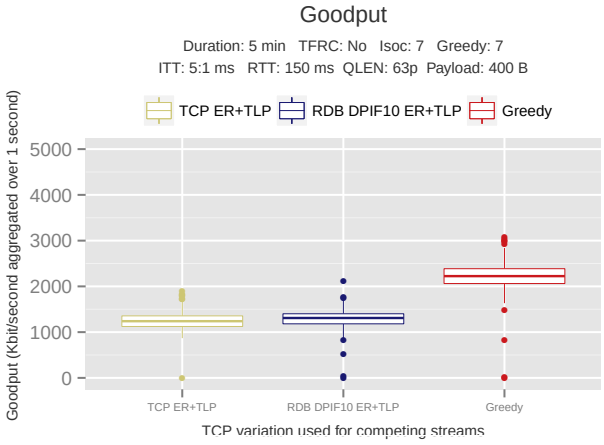
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
376	300	302	312	482	622	714	777	787	813	863	901	1225	2135	3049
373	300	302	307	479	608	692	765	782	796	850	893	1329	1470	1834
365	300	302	308	535	598	605	620	648	709	784	850	896	1728	1747

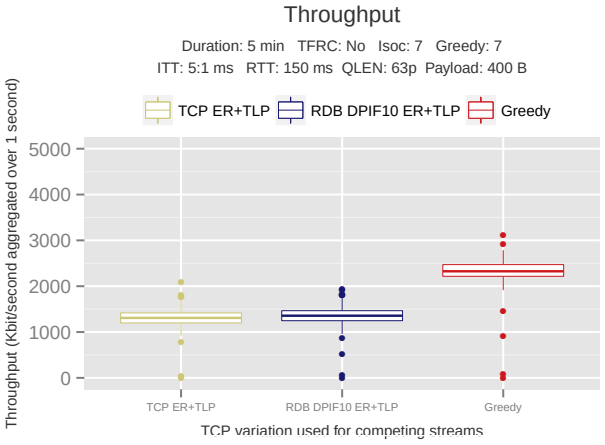
(c)

Figure A.4.19



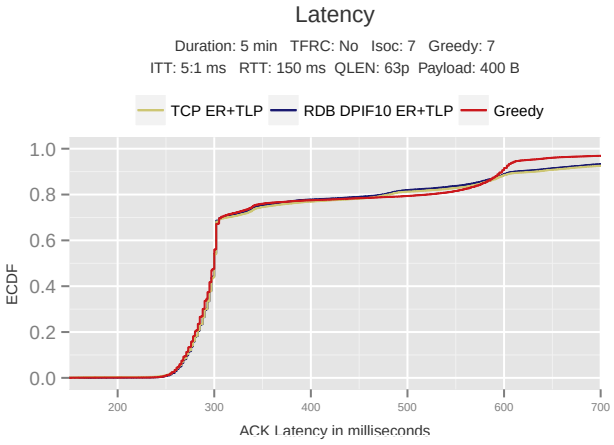
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	60 / 61 / 64	1438/1442/1444	5/1/4/5/6/22
RDB DPIF10 ER+TLP	rdbsender	7	57 / 59 / 62	1444/1445/1445	5/1/4/5/7/21
Greedy	zsender	7	33 / 35 / 37	1443/1445/1447	9/1/7/8/10/22

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
34.3 K	1576	1580	0	49.5 M	47.3 M	2.2 M	2.2 M	4.53 / 4.59	7231
35.1 K	1480	1478	105	50.7 M	48.5 M	2.2 M	2.1 M	4.18 / 4.22	7220
60.5 K	2052	2056	0	87.5 M	84.5 M	3.0 M	3.0 M	3.39 / 3.39	13424

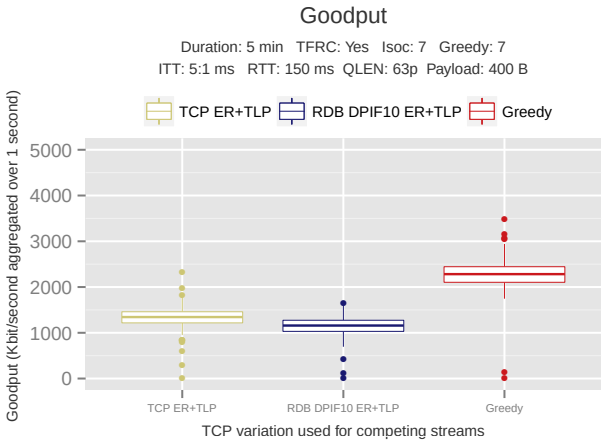
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
379	300	302	314	482	632	714	772	784	801	859	956	1810	4358	4417
374	300	302	309	474	612	690	763	782	796	852	903	1437	2105	2400
365	300	302	307	520	598	605	622	649	709	788	848	1107	2008	2256

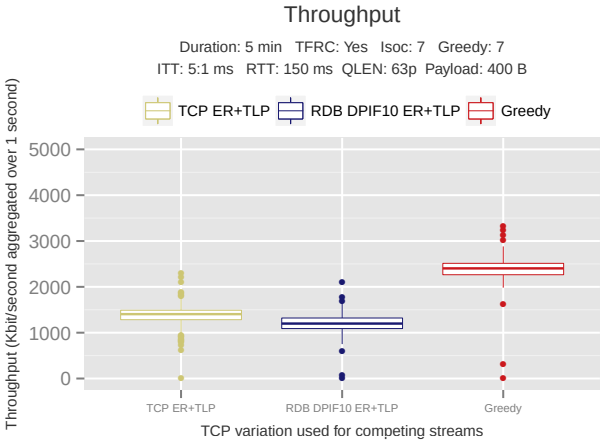
(c)

Figure A.4.20



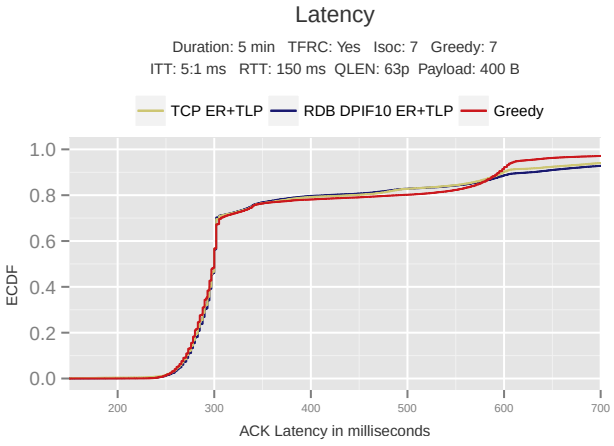
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	56 / 58 / 59	1444/1444/1445	5/1/4/5/7/14
RDB DPIF10 ER+TLP	rdbsender	7	60 / 67 / 71	1443/1444/1445	5/1/3/4/6/30
Greedy	zsender	7	31 / 34 / 36	1443/1446/1447	9/2/7/9/10/26

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
36.2 K	1432	1439	0	52.4 M	50.3 M	2.0 M	2.0 M	3.88 / 3.95	7019
31.2 K	1299	1294	87	45.1 M	43.1 M	1.9 M	1.8 M	4.13 / 4.16	5769
61.9 K	1942	1946	0	89.5 M	86.7 M	2.8 M	2.8 M	3.13 / 3.14	13434

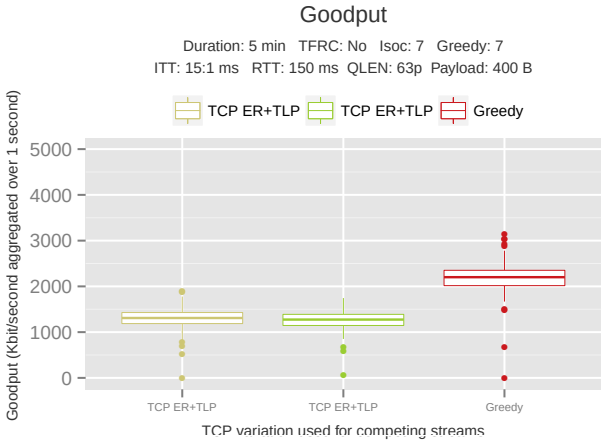
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
369	300	302	305	443	600	670	748	775	789	833	882	3227	3861	4241
377	300	302	302	416	627	709	775	792	838	869	943	1983	12283	12550
361	300	302	307	491	595	603	617	641	692	762	835	892	1059	1151

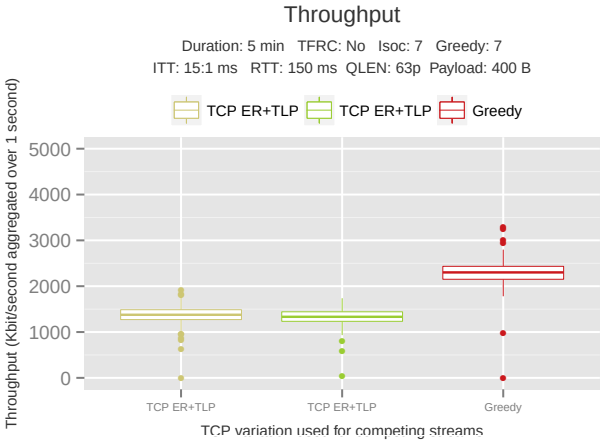
(c)

Figure A.4.21



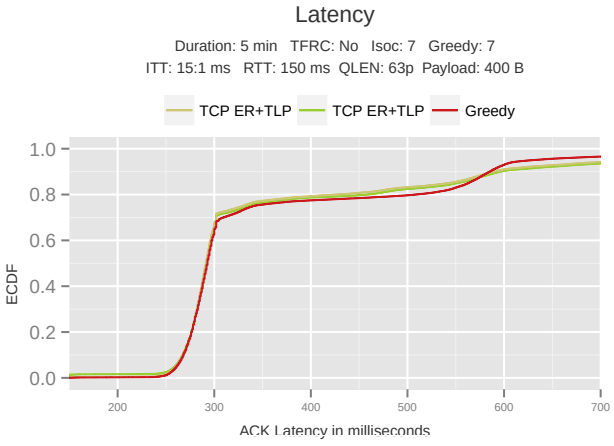
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	51 / 54 / 59	1250/1314/1370	6/1/4/6/8/20
TCP ER+TLP	rdbsender	7	53 / 56 / 61	1303/1345/1376	6/1/4/5/7/20
Greedy	zsender	7	33 / 35 / 39	1444/1446/1447	9/1/7/8/10/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
39.1 K	1563	1567	0	51.3 M	49.4 M	1.9 M	1.9 M	3.80 / 4.00	7459
36.9 K	1574	1574	0	49.5 M	47.5 M	2.0 M	2.0 M	4.11 / 4.27	7118
59.6 K	2232	2236	0	86.2 M	83.0 M	3.2 M	3.2 M	3.74 / 3.74	12873

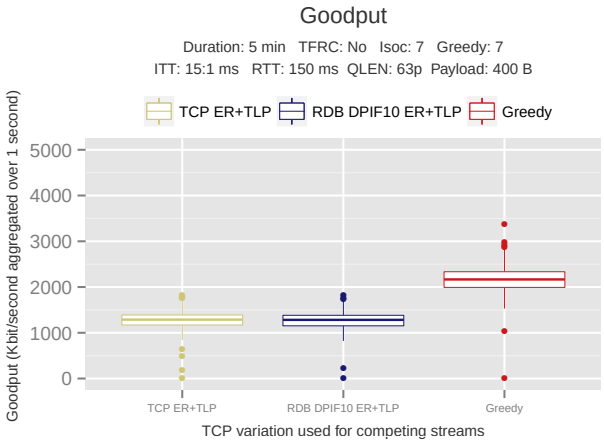
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
360	292	296	302	426	589	657	731	762	783	831	884	1357	2007	3295
363	292	297	302	458	596	676	755	773	804	850	937	1434	1803	1942
363	293	298	310	508	585	600	629	669	730	801	844	1416	4969	8670

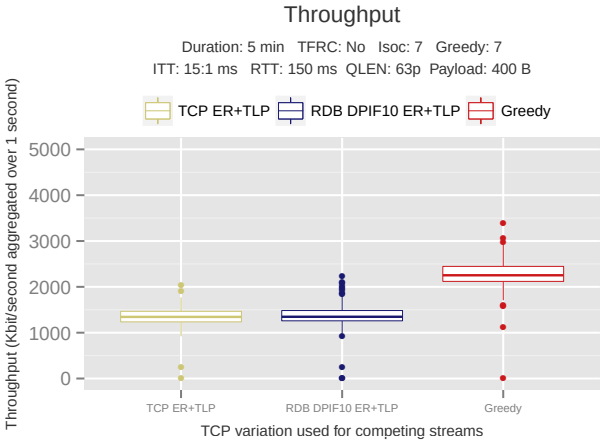
(c)

Figure A.4.22



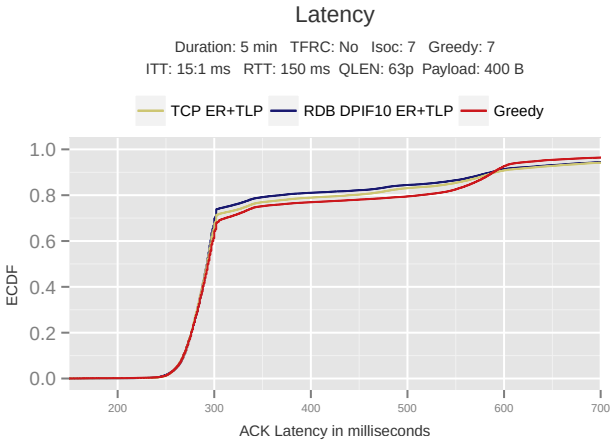
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	51 / 55 / 61	1273/1317/1391	6/1/4/6/7/22
RDB DPIF10 ER+TLP	rdbsender	7	50 / 57 / 61	1335/1389/1419	6/1/4/6/7/22
Greedy	zsender	7	34 / 36 / 38	1446/1447/1447	8/1/7/8/10/26

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
38.3 K	1539	1543	0	50.4 M	48.4 M	1.9 M	1.9 M	3.84 / 4.02	7446
36.9 K	1495	1352	3.0 K	51.2 M	47.3 M	3.9 M	1.9 M	4.05 / 4.05	6488
59.0 K	2179	2183	0	85.4 M	82.3 M	3.2 M	3.2 M	3.69 / 3.69	13191

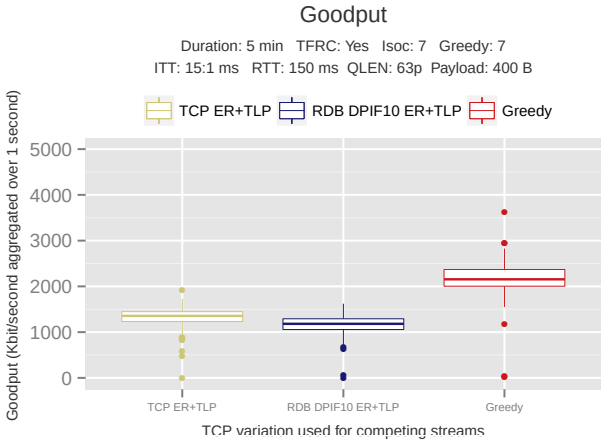
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
363	293	297	302	440	591	657	736	763	783	830	882	1627	3925	3978
355	293	296	301	368	586	646	722	757	779	821	880	1415	2035	2776
362	294	298	312	513	589	603	637	675	732	793	840	1107	2185	2470

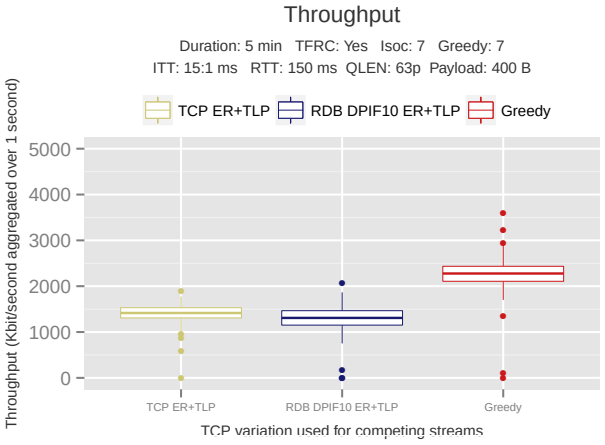
(c)

Figure A.4.23



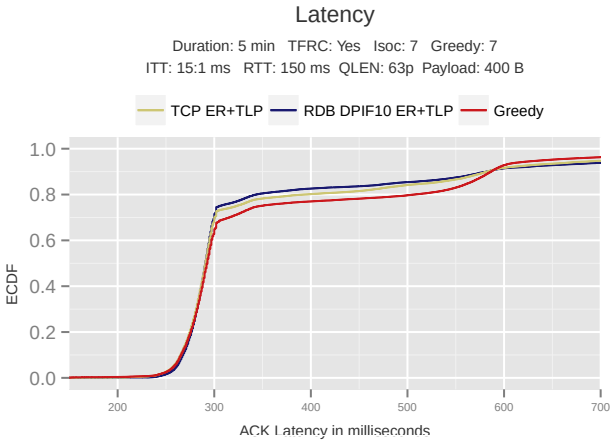
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	48 / 51 / 54	1214/1279/1334	6/1/4/6/8/22
RDB DPIF10 ER+TLP	rdbsender	7	50 / 60 / 72	1325/1395/1441	7/1/4/5/8/42
Greedy	zsender	7	35 / 35 / 36	1445/1446/1447	9/1/7/8/10/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.9 K	1512	1516	0	52.3 M	50.5 M	1.8 M	1.8 M	3.46 / 3.69	7659
35.1 K	1464	1236	4.4 K	48.8 M	43.9 M	4.9 M	1.7 M	4.15 / 4.16	5466
59.0 K	2234	2238	0	85.4 M	82.1 M	3.2 M	3.2 M	3.78 / 3.78	12815

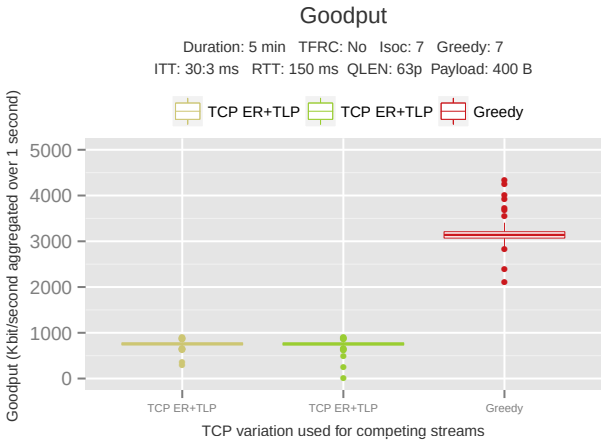
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
355	291	295	300	393	583	633	712	751	775	817	881	1342	1775	1903
356	291	295	300	343	582	658	757	781	823	859	1060	1646	2940	3143
363	293	297	315	507	586	602	642	684	742	802	849	1357	5937	6743

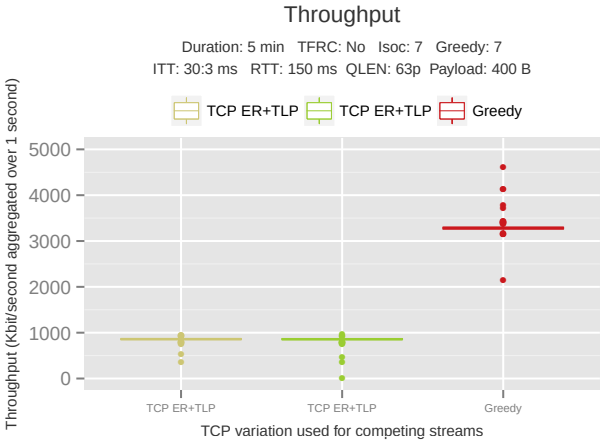
(c)

Figure A.4.24



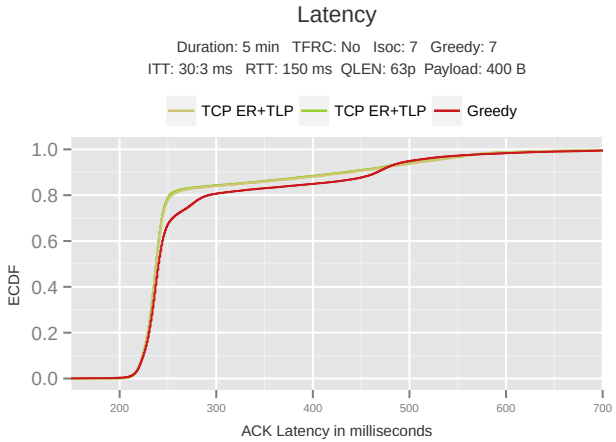
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	35 / 36 / 37	479/496/517	9/1/6/8/11/14
TCP ER+TLP	rdbsender	7	35 / 36 / 37	485/497/515	9/1/6/8/11/13
Greedy	zsender	7	24 / 25 / 27	1446/1447/1447	10/2/8/9/11/39

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
58.5 K	1360	1360	0	29.0 M	28.4 M	617 K	617 K	2.13 / 2.32	9074
57.6 K	1282	1286	0	28.6 M	28.0 M	582 K	582 K	2.04 / 2.22	8836
84.5 K	2296	2296	0	122 M	119 M	3.3 M	3.3 M	2.72 / 2.72	15193

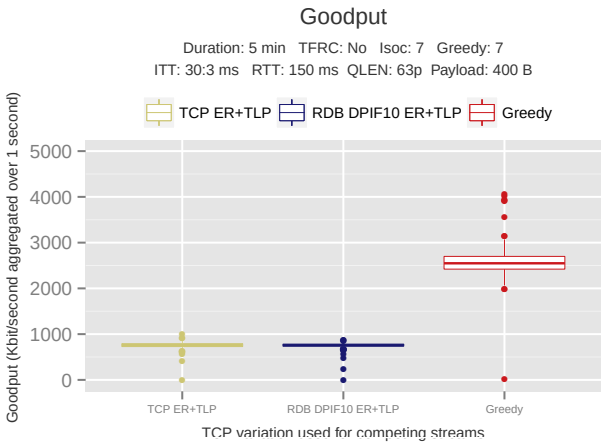
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
275	238	241	244	254	439	488	523	539	557	573	615	838	1167	1433
274	238	240	244	252	432	482	519	536	555	572	617	865	1249	1813
285	240	244	255	292	466	482	501	519	542	581	648	880	1151	1294

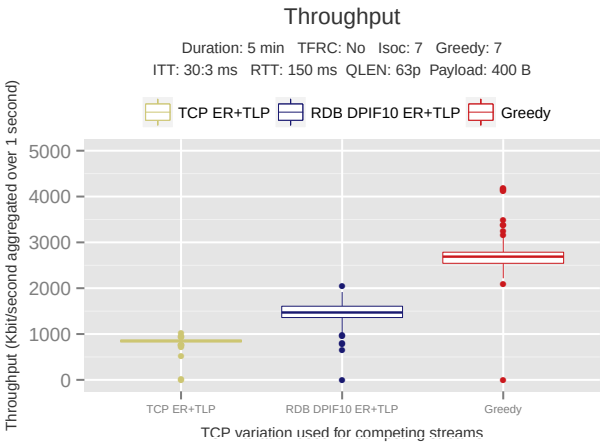
(c)

Figure A.4.25



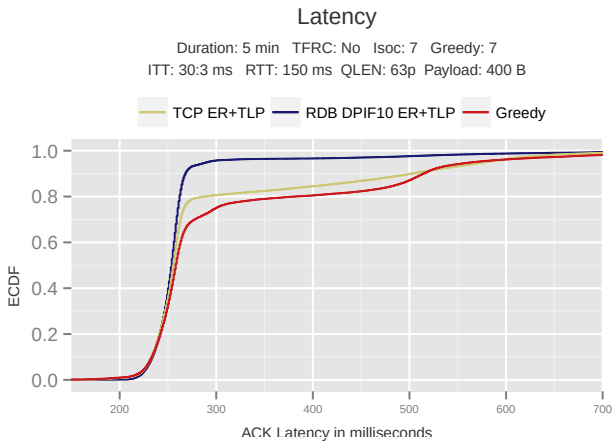
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	38 / 40 / 41	528/550/566	8/1/6/8/10/14
RDB DPIF10 ER+TLP	rdbsender	7	40 / 44 / 46	1112/1122/1127	7/1/5/7/9/14
Greedy	zsender	7	28 / 30 / 32	1447/1447/1447	9/2/7/8/10/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
53.1 K	1544	1548	0	29.2 M	28.5 M	753 K	753 K	2.57 / 2.90	9230
47.5 K	1508	361	34.4 K	53.4 M	28.0 M	25.3 M	343 K	3.15 / 3.17	1556
69.8 K	2531	2535	0	101 M	97.3 M	3.7 M	3.7 M	3.62 / 3.63	13933

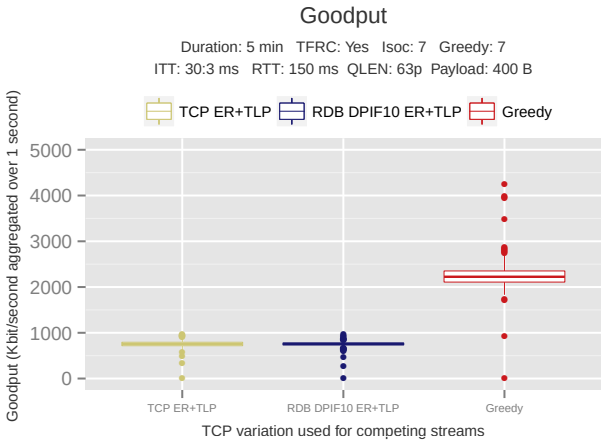
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
300	256	259	263	288	502	547	579	594	611	639	695	1017	1972	2741
264	254	256	259	262	268	274	291	311	448	526	640	1019	1608	2082
314	258	263	278	382	515	534	566	593	635	690	747	1124	1568	1574

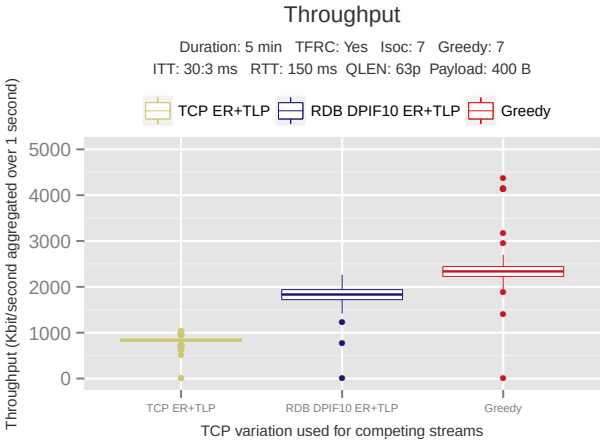
(c)

Figure A.4.26



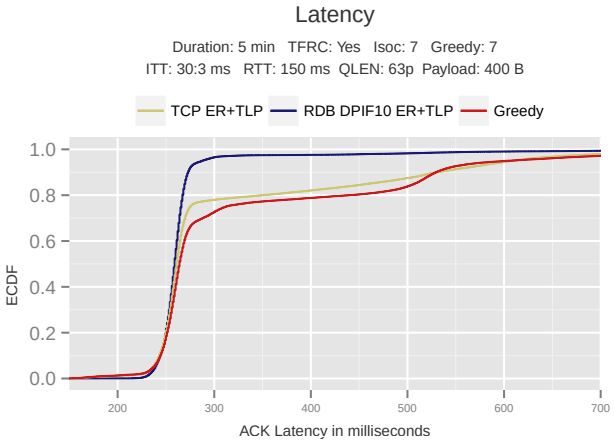
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	7	43 / 44 / 46	598/618/651	7/1/5/7/9/14
RDB DPIF10 ER+TLP	rdbsender	7	35 / 36 / 37	1150/1158/1165	9/1/7/9/11/26
Greedy	zsender	7	32 / 34 / 37	1445/1447/1447	8/1/6/8/9/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
47.5 K	1594	1598	0	29.3 M	28.5 M	893 K	893 K	3.04 / 3.35	9137
57.3 K	1945	314	50.3 K	66.3 M	28.0 M	38.3 M	294 K	3.43 / 3.39	1275
61.4 K	2528	2532	0	88.8 M	85.1 M	3.7 M	3.7 M	4.11 / 4.12	12853

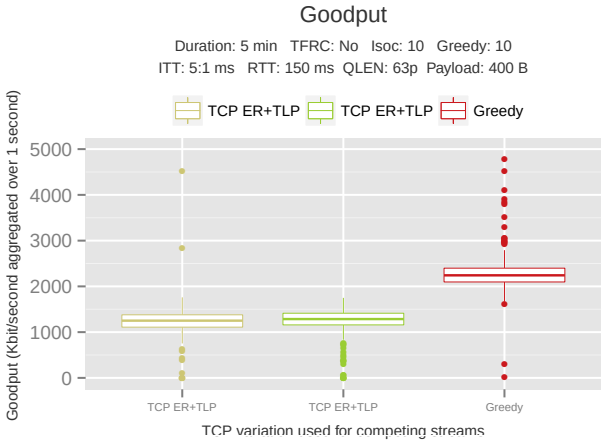
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
317	261	265	270	350	530	577	608	627	656	702	778	1253	1852	2235
268	259	261	264	267	273	277	288	295	311	466	587	1069	1975	3136
332	264	269	288	440	530	555	604	644	691	739	778	2201	8563	9045

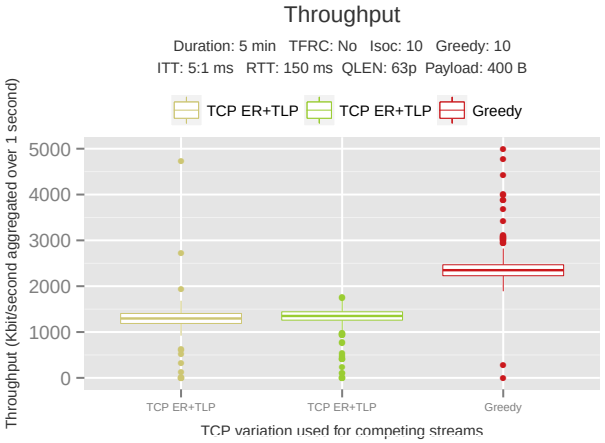
(c)

Figure A.4.27



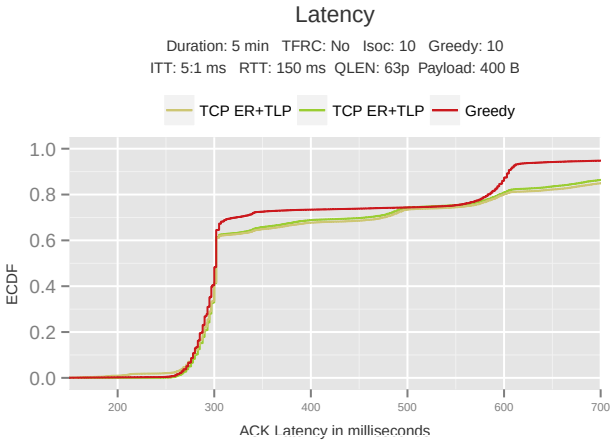
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	82 / 87 / 94	1439/1444/1446	4/1/3/4/5/13
TCP ER+TLP	rdbsender	10	84 / 85 / 89	1441/1444/1446	4/1/3/4/5/11
Greedy	zsender	10	43 / 47 / 54	1445/1446/1446	7/2/5/6/7/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
34.8 K	2480	2481	0	50.2 M	46.7 M	3.5 M	3.5 M	7.01 / 7.13	9619
35.0 K	2385	2389	0	50.5 M	47.1 M	3.4 M	3.4 M	6.72 / 6.82	9634
63.4 K	2916	2916	0	91.6 M	87.4 M	4.2 M	4.2 M	4.60 / 4.60	16043

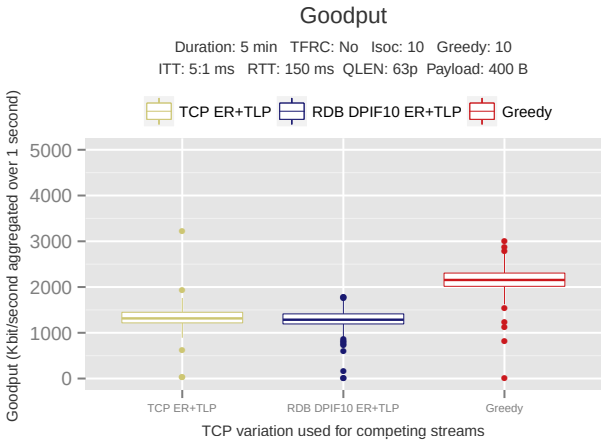
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
438	302	302	474	600	792	859	886	898	1070	1132	1186	2669	8686	21100
431	302	302	457	595	784	821	881	892	999	1099	1177	3150	10381	18090
387	302	302	323	583	605	612	724	813	857	881	893	1130	1399	1506

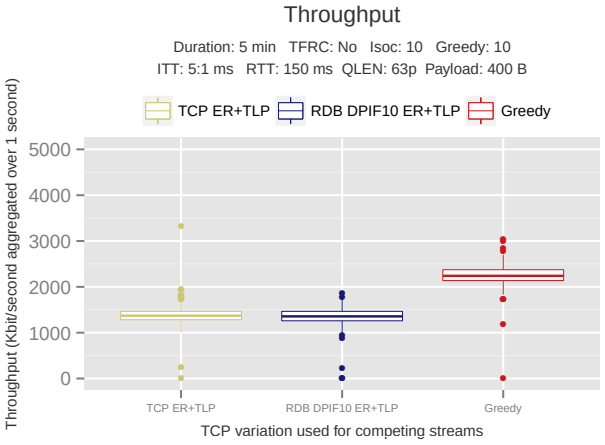
(c)

Figure A.4.28



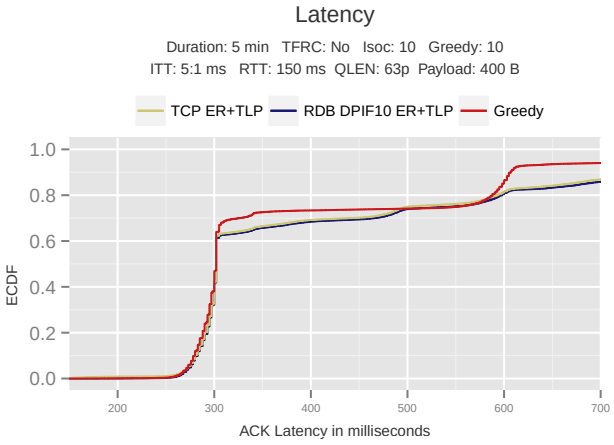
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	78 / 81 / 84	1443/1443/1444	4/1/3/4/5/15
RDB DPIF10 ER+TLP	rdbsender	10	78 / 83 / 86	1444/1445/1446	4/1/3/4/5/13
Greedy	zsender	10	45 / 51 / 57	1441/1445/1447	6/2/5/6/7/39

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
37.3 K	2458	2467	0	53.9 M	50.4 M	3.5 M	3.5 M	6.56 / 6.58	10044
36.2 K	2498	2480	90	52.3 M	48.7 M	3.6 M	3.6 M	6.89 / 6.90	9838
59.1 K	2827	2834	0	85.4 M	81.3 M	4.1 M	4.1 M	4.77 / 4.78	14880

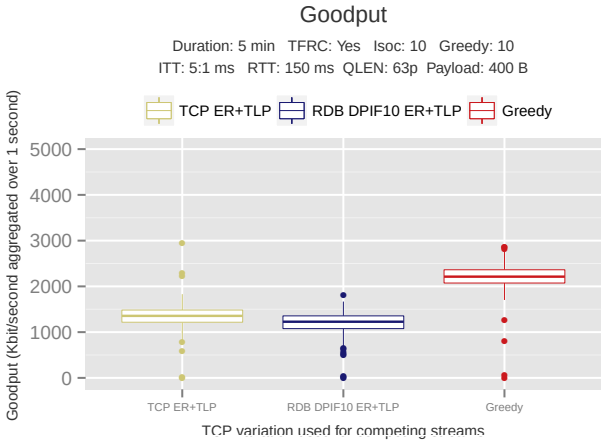
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
421	302	302	440	593	779	799	872	886	901	1085	1172	1475	2000	2312
429	302	302	465	598	787	847	884	896	1020	1111	1182	1632	1965	2296
391	302	302	327	588	607	624	790	849	874	888	898	1192	1787	2364

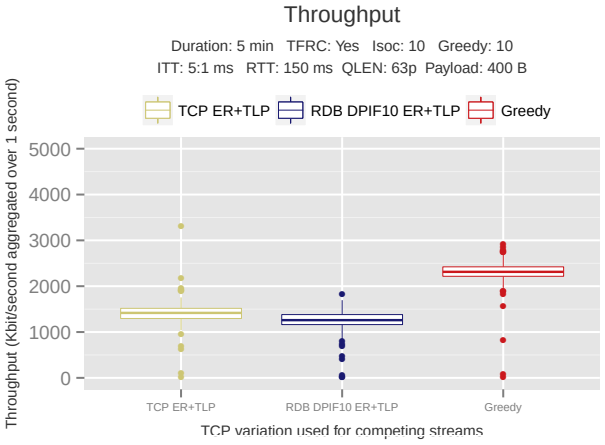
(c)

Figure A.4.29



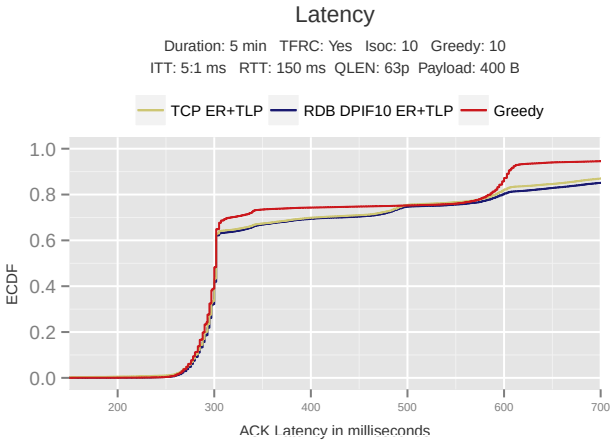
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	75 / 80 / 85	1442/1443/1444	4/1/3/4/5/15
RDB DPIF10 ER+TLP	rdbsender	10	81 / 89 / 96	1443/1444/1446	4/1/3/4/5/24
Greedy	zsender	10	47 / 50 / 52	1445/1446/1447	6/2/5/6/7/22

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
37.9 K	2445	2457	0	54.7 M	51.2 M	3.5 M	3.5 M	6.39 / 6.44	9879
33.8 K	2347	2344	90	48.8 M	45.4 M	3.4 M	3.3 M	6.92 / 6.94	8736
60.6 K	2810	2817	0	87.7 M	83.7 M	4.1 M	4.1 M	4.62 / 4.63	14821

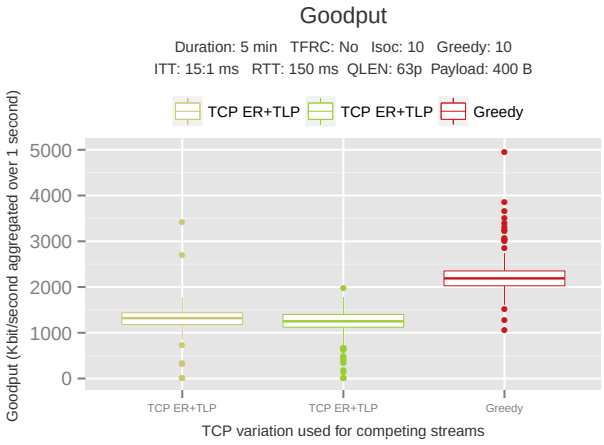
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
419	302	302	404	591	779	801	876	891	906	1090	1179	1494	2487	3273
449	302	302	431	598	792	867	893	908	1084	1160	1199	4627	22202	22230
386	302	302	317	586	605	615	748	825	864	884	896	1177	1465	1643

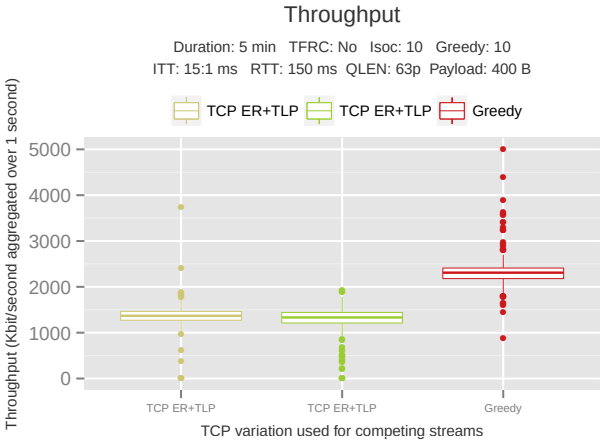
(c)

Figure A.4.30



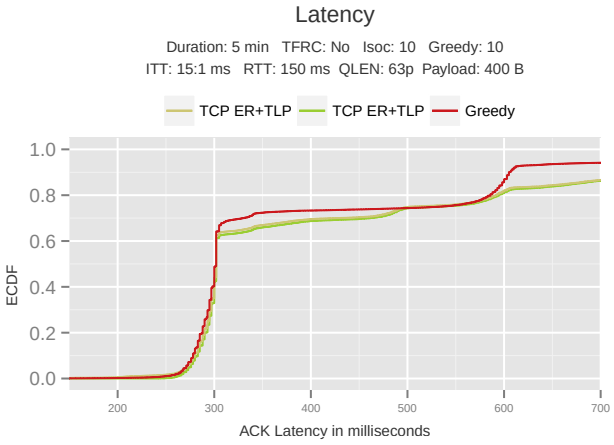
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	78 / 82 / 85	1420/1439/1446	4/1/3/4/5/19
TCP ER+TLP	rdbsender	10	80 / 87 / 96	1433/1443/1446	4/1/3/4/5/13
Greedy	zsender	10	42 / 49 / 60	1445/1445/1446	7/1/5/6/7/43

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
36.9 K	2422	2429	0	53.2 M	49.7 M	3.5 M	3.5 M	6.49 / 6.55	9712
34.5 K	2332	2336	0	49.8 M	46.5 M	3.3 M	3.3 M	6.69 / 6.76	9346
61.6 K	2926	2926	0	89.1 M	84.9 M	4.2 M	4.2 M	4.74 / 4.75	15429

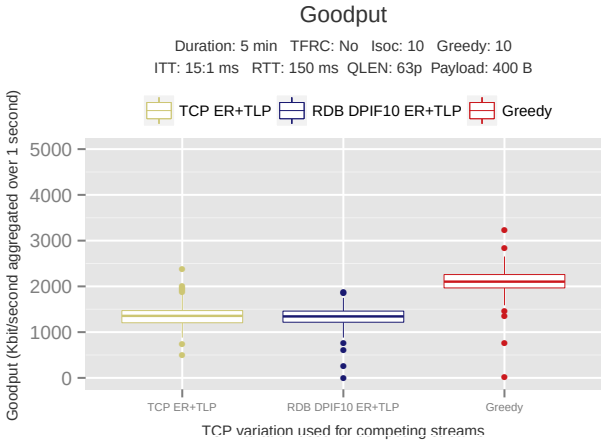
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
423	302	302	423	591	784	821	879	889	925	1097	1179	1769	5113	7873
435	302	302	462	595	787	831	881	892	984	1107	1184	3168	15359	36710
389	302	302	330	586	605	624	767	833	867	886	896	1199	2320	2326

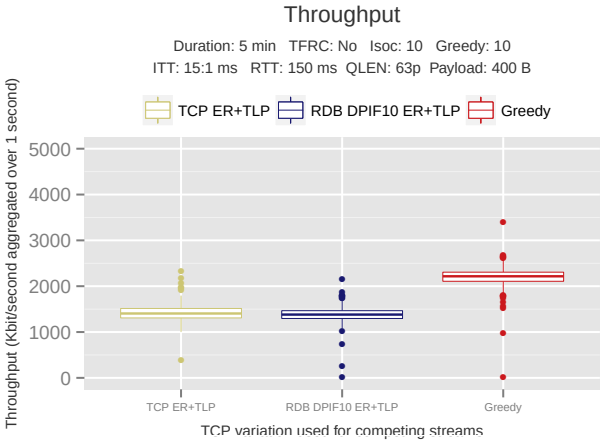
(c)

Figure A.4.31



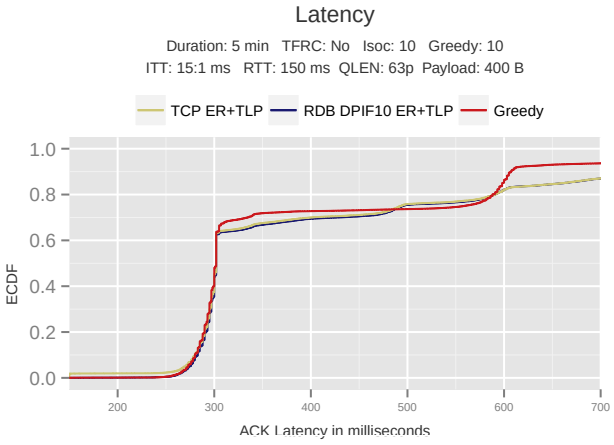
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	71 / 77 / 82	1367/1408/1418	4/1/3/4/5/21
RDB DPIF10 ER+TLP	rdbsender	10	77 / 81 / 85	1428/1439/1443	4/1/3/4/5/20
Greedy	zsender	10	48 / 52 / 56	1440/1444/1447	6/1/5/6/7/18

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
38.9 K	2553	2553	0	54.8 M	51.3 M	3.5 M	3.5 M	6.47 / 6.56	10094
37.0 K	2468	2404	359	53.3 M	49.6 M	3.7 M	3.4 M	6.62 / 6.66	9859
57.8 K	2871	2878	0	83.5 M	79.4 M	4.1 M	4.1 M	4.96 / 4.96	14656

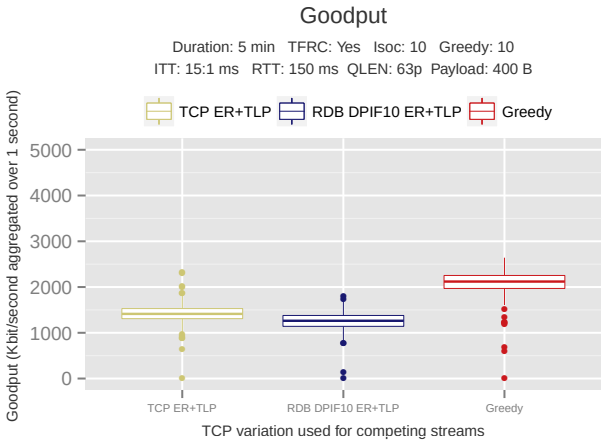
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
416	302	302	400	591	778	797	874	890	926	1085	1179	1487	1790	1991
421	302	302	428	591	779	797	871	886	903	1090	1177	1662	3512	3597
398	302	302	334	588	607	649	819	858	879	889	898	2469	6627	10290

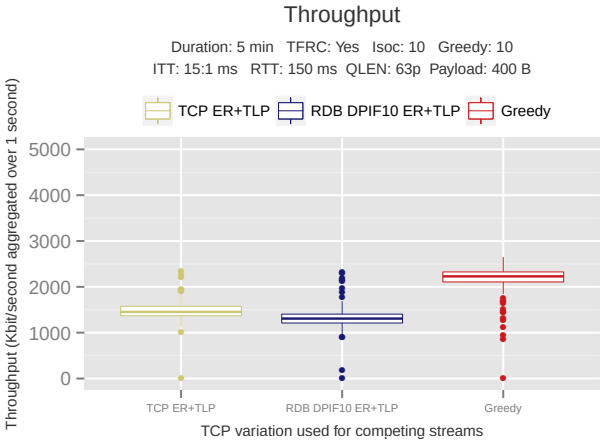
(c)

Figure A.4.32



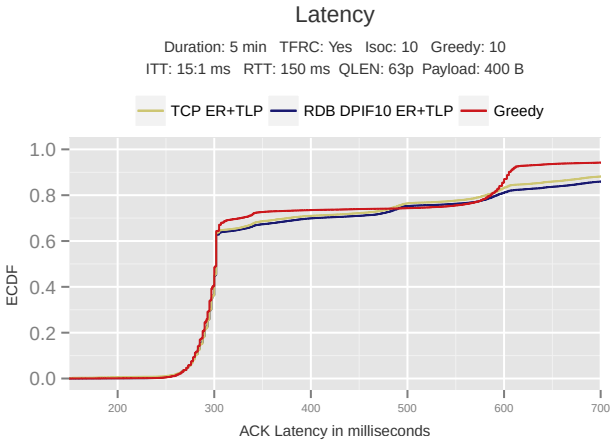
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	73 / 76 / 80	1421/1431/1439	4/1/3/4/5/20
RDB DPIF10 ER+TLP	rdbsender	10	75 / 84 / 91	1410/1436/1445	4/1/3/4/5/62
Greedy	zsender	10	48 / 53 / 68	1439/1445/1447	6/1/5/6/7/19

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
39.7 K	2496	2467	0	56.8 M	53.4 M	3.4 M	3.4 M	6.09 / 6.29	10064
35.5 K	2466	2352	1.0 K	51.0 M	46.9 M	4.1 M	3.3 M	6.90 / 6.93	9166
57.6 K	2752	2759	0	83.3 M	79.4 M	4.0 M	4.0 M	4.76 / 4.77	14410

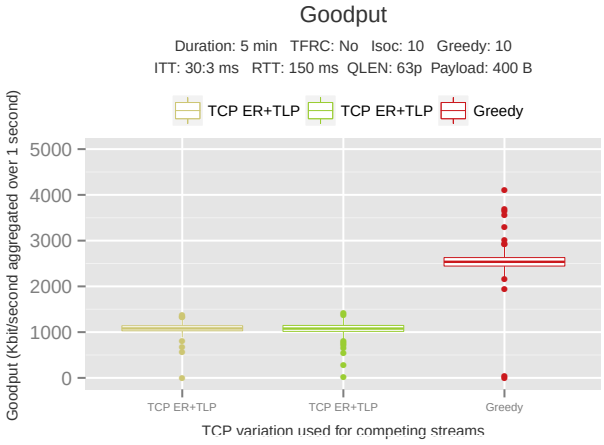
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
413	302	302	375	581	765	792	848	879	896	1068	1170	2182	3819	3878
425	302	302	404	593	789	852	886	896	1028	1111	1186	1753	3174	3306
400	302	302	327	586	605	627	777	839	869	886	898	1204	51530	56130

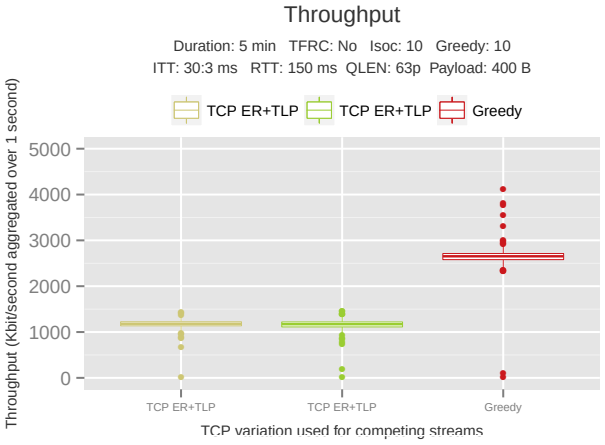
(c)

Figure A.4.33



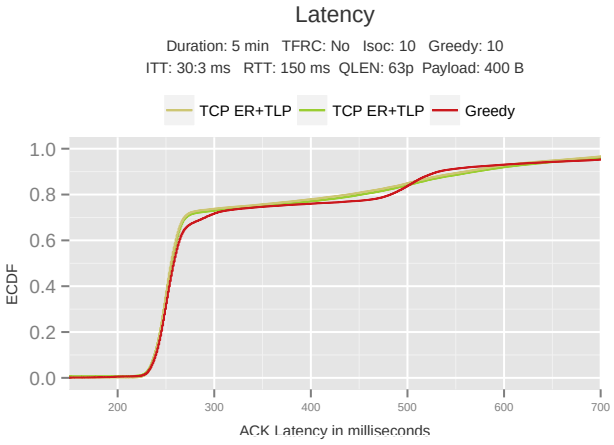
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	48 / 51 / 57	676/729/817	6/1/4/6/8/14
TCP ER+TLP	rdbsender	10	50 / 53 / 59	710/762/848	6/1/4/6/7/13
Greedy	zsender	10	40 / 43 / 46	1445/1446/1447	7/1/5/6/8/24

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
58.4 K	2745	2752	0	42.5 M	40.6 M	1.8 M	1.8 M	4.34 / 4.70	12791
55.5 K	2801	2800	0	42.2 M	40.2 M	2.0 M	2.0 M	4.78 / 5.04	12352
70.2 K	3636	3643	0	102 M	96.3 M	5.3 M	5.3 M	5.17 / 5.18	15621

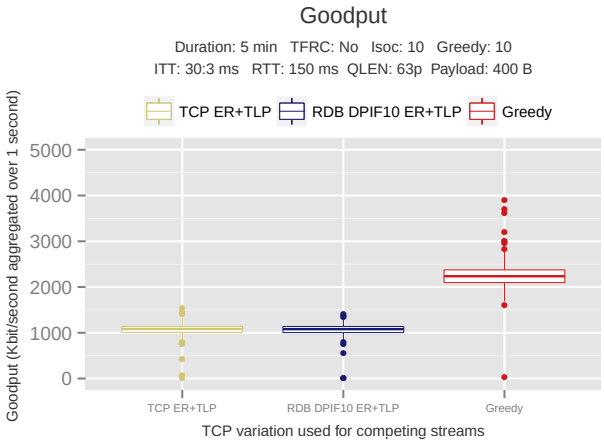
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
329	255	260	269	438	560	604	654	684	710	751	906	1440	1875	2193
335	256	261	271	452	570	622	676	700	728	786	991	1555	2402	2837
336	258	264	291	483	532	599	691	722	742	767	949	1510	3837	4497

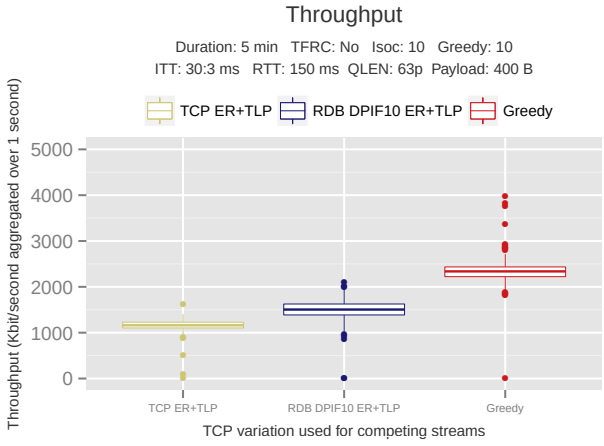
(c)

Figure A.4.34



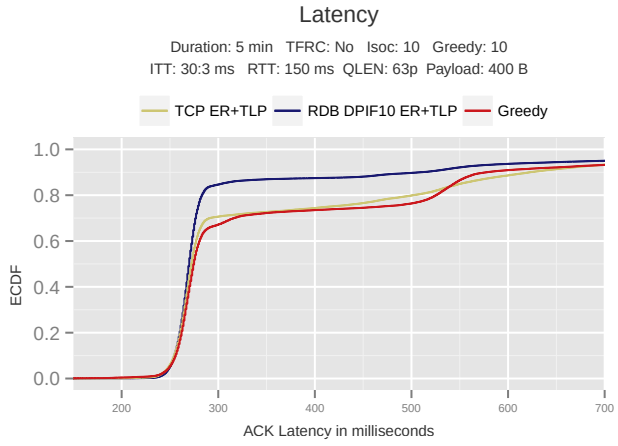
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	53 / 59 / 62	745/845/893	6/1/4/5/7/14
RDB DPIF10 ER+TLP	rdbsender	10	58 / 62 / 69	1150/1175/1205	6/1/4/5/7/14
Greedy	zsender	10	44 / 49 / 58	1445/1446/1447	6/1/5/6/7/26

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
50.7 K	2781	2788	0	42.8 M	40.6 M	2.1 M	2.1 M	4.94 / 5.48	11685
47.6 K	2621	1476	20.7 K	55.9 M	39.9 M	16.1 M	1.7 M	5.48 / 5.50	5023
62.2 K	3511	3518	0	89.9 M	84.9 M	5.1 M	5.1 M	5.64 / 5.64	14395

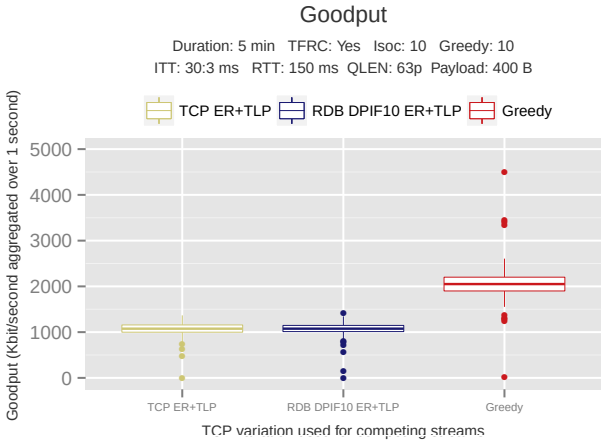
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
363	272	277	291	502	623	690	733	752	799	879	1080	1661	2096	2318
322	270	273	277	283	509	570	697	735	772	832	1078	1683	2523	5289
369	275	281	316	525	578	689	769	790	806	828	1045	2052	7069	15320

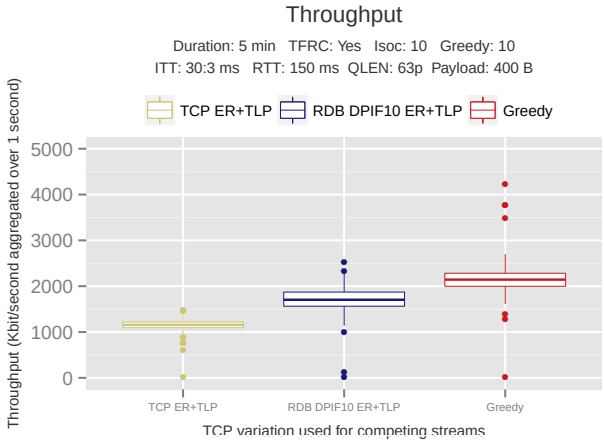
(c)

Figure A.4.35



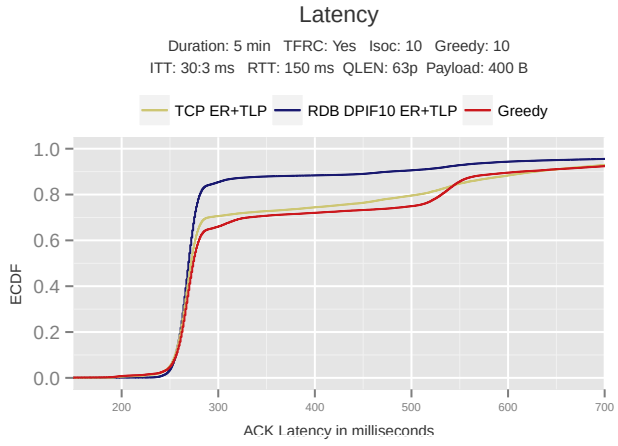
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	10	58 / 60 / 62	811/851/893	6/1/4/5/7/14
RDB DPIF10 ER+TLP	rdbsender	10	52 / 54 / 56	1139/1161/1176	6/1/5/6/8/47
Greedy	zsender	10	46 / 53 / 56	1446/1446/1447	6/1/4/5/7/27

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
50.0 K	2747	2756	0	42.6 M	40.4 M	2.1 M	2.1 M	4.97 / 5.49	11459
55.0 K	3302	1518	31.2 K	63.8 M	40.0 M	23.8 M	1.7 M	5.98 / 6.00	5404
57.2 K	3510	3516	0	82.8 M	77.7 M	5.1 M	5.1 M	6.13 / 6.13	13704

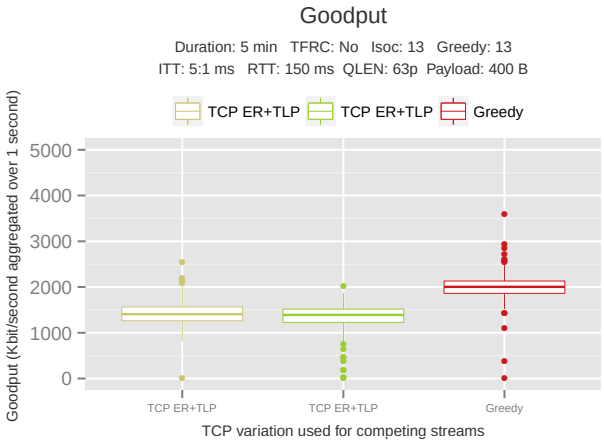
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
366	272	276	290	507	628	701	740	771	811	941	1086	2075	4702	7519
318	269	272	275	281	477	554	646	722	753	815	1058	1651	2426	2720
383	274	280	331	531	613	725	784	799	815	846	1087	4321	14514	14770

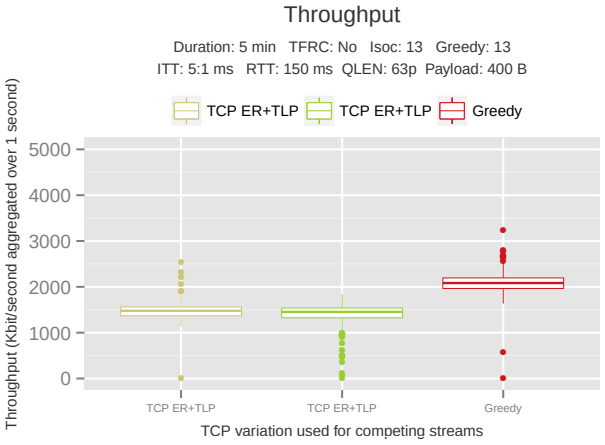
(c)

Figure A.4.36



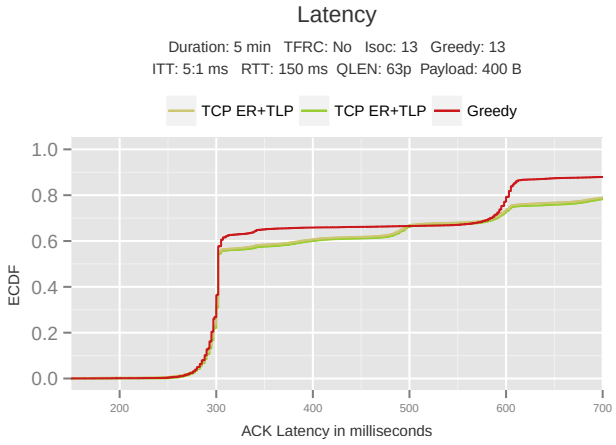
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	91 / 96 / 101	1440/1441/1442	4/1/3/3/4/15
TCP ER+TLP	rdbsender	13	94 / 100 / 109	1439/1440/1442	3/1/3/3/4/13
Greedy	zsender	13	63 / 70 / 91	1444/1446/1447	5/1/4/5/6/26

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.8 K	3605	3615	0	58.9 M	53.7 M	5.2 M	5.2 M	8.76 / 8.82	12807
38.8 K	3540	3555	0	56.0 M	50.9 M	5.0 M	5.0 M	8.98 / 9.11	12362
56.2 K	3711	3721	0	81.4 M	76.0 M	5.4 M	5.4 M	6.58 / 6.59	16447

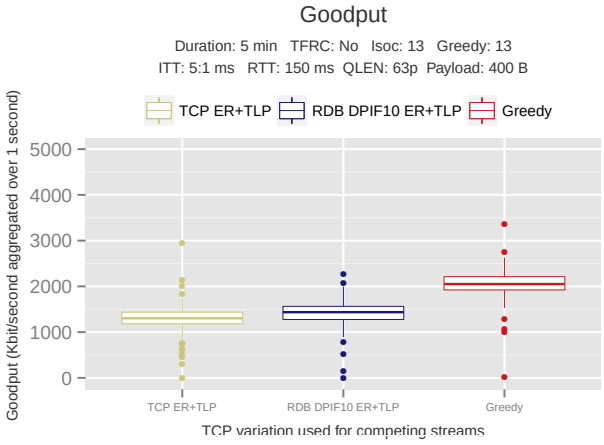
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
483	302	387	587	721	891	906	1094	1122	1186	1196	1208	1804	3043	3740
514	302	399	591	762	893	1002	1104	1167	1191	1201	1400	6875	30535	30630
440	302	305	586	603	857	891	898	901	903	908	1186	1499	3235	4968

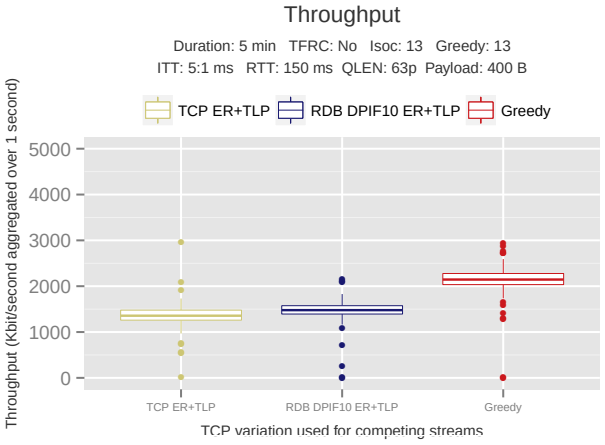
(c)

Figure A.4.37



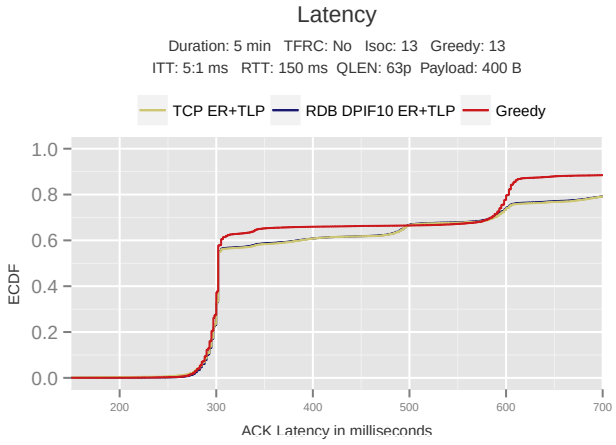
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	89 / 114 / 295	1417/1439/1442	3/1/3/3/4/12
RDB DPIF10 ER+TLP	rdbsender	13	91 / 96 / 106	1442/1444/1444	4/1/3/3/4/15
Greedy	zsender	13	62 / 68 / 75	1436/1444/1446	5/1/4/5/6/22

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
37.6 K	3405	3415	0	54.2 M	49.4 M	4.8 M	4.8 M	8.90 / 9.05	11714
40.5 K	3595	3593	117	58.4 M	53.2 M	5.2 M	5.1 M	8.86 / 8.88	12663
57.7 K	3834	3845	0	83.5 M	77.9 M	5.5 M	5.5 M	6.62 / 6.63	16761

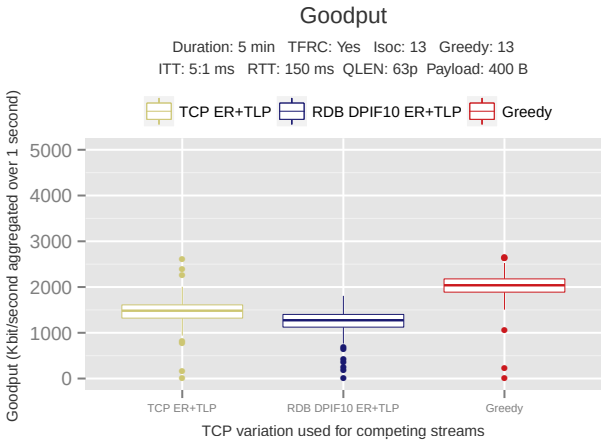
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
567	302	387	588	719	893	908	1102	1167	1192	1201	1398	4976	203987	204400
490	302	385	586	721	891	905	1094	1116	1184	1196	1206	2371	21089	22360
439	302	305	586	603	845	888	896	898	903	905	1184	1507	4611	6951

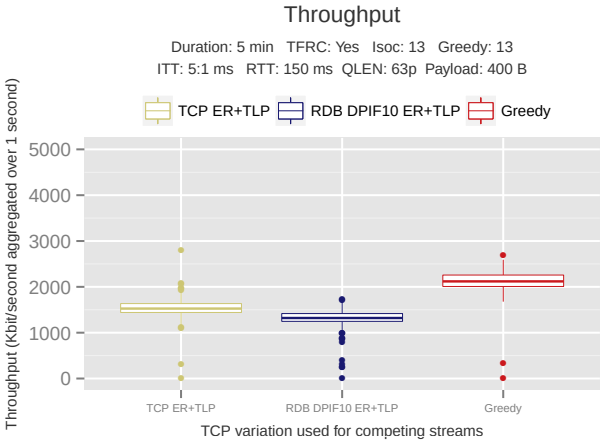
(c)

Figure A.4.38



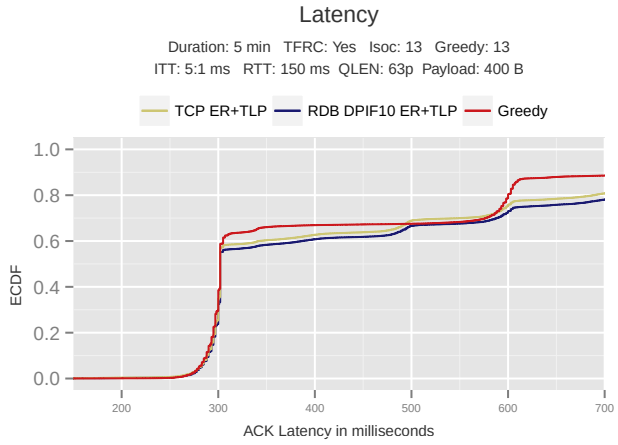
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	89 / 93 / 100	1441/1442/1443	4/1/3/3/4/15
RDB DPIF10 ER+TLP	rdbsender	13	98 / 114 / 261	1434/1443/1446	3/1/2/3/4/34
Greedy	zsender	13	61 / 69 / 88	1435/1444/1447	5/1/4/5/6/23

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
42.2 K	3519	3529	0	60.9 M	55.9 M	5.0 M	5.0 M	8.27 / 8.33	12711
36.4 K	3374	3354	117	52.5 M	47.7 M	4.9 M	4.8 M	9.23 / 9.27	11291
56.9 K	3670	3680	0	82.3 M	77.0 M	5.3 M	5.3 M	6.43 / 6.44	16154

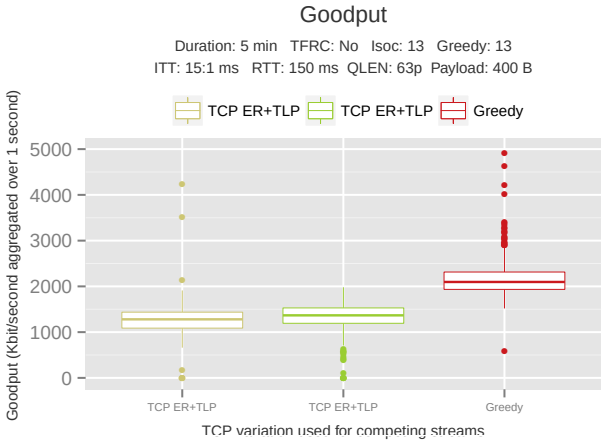
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
468	302	342	550	690	874	898	1031	1098	1158	1189	1203	1796	4142	9222
528	302	387	588	764	896	1024	1111	1175	1194	1204	1485	8129	25047	163400
442	302	305	576	600	834	886	896	898	903	908	1184	1711	11237	65610

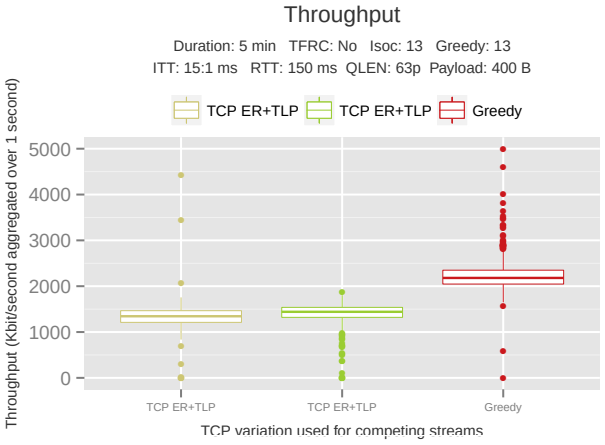
(c)

Figure A.4.39



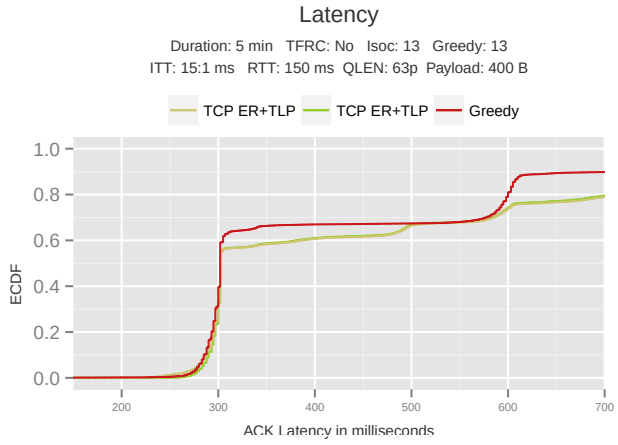
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	97 / 116 / 276	1437/1443/1446	3/1/3/3/4/13
TCP ER+TLP	rdbsender	13	91 / 104 / 115	1439/1444/1446	3/1/3/3/4/11
Greedy	zsender	13	56 / 64 / 70	1445/1446/1446	5/1/4/5/6/28

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
36.6 K	3212	3216	0	52.9 M	48.3 M	4.6 M	4.6 M	8.70 / 8.77	11533
37.6 K	3282	3278	0	54.4 M	49.7 M	4.7 M	4.7 M	8.60 / 8.70	11893
61.5 K	3966	3966	0	88.9 M	83.2 M	5.7 M	5.7 M	6.45 / 6.45	17679

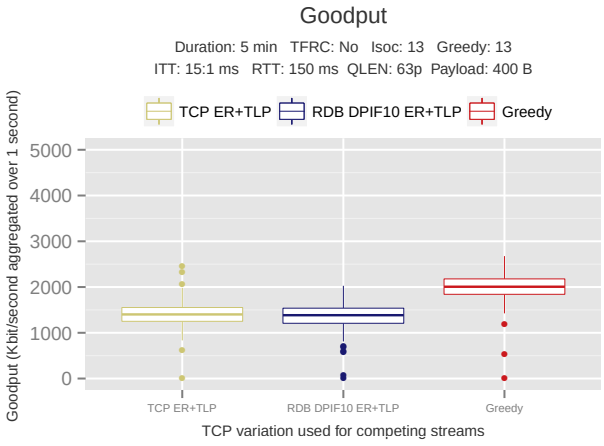
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
500	302	387	586	736	889	905	1092	1116	1179	1194	1206	2265	36702	186200
491	302	382	583	712	886	903	1087	1106	1179	1194	1206	3548	33665	39840
428	302	305	578	600	735	881	893	896	900	903	1155	1487	1635	3289

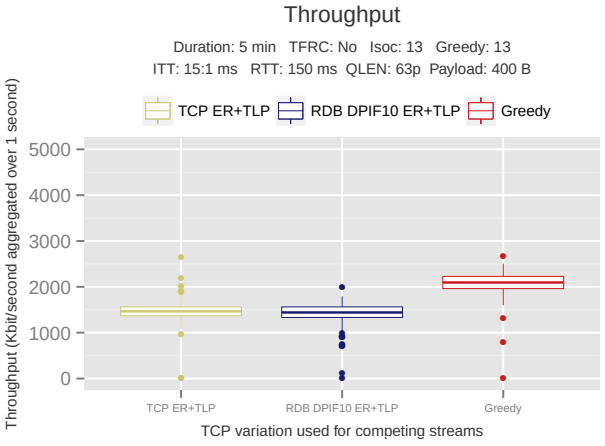
(c)

Figure A.4.40



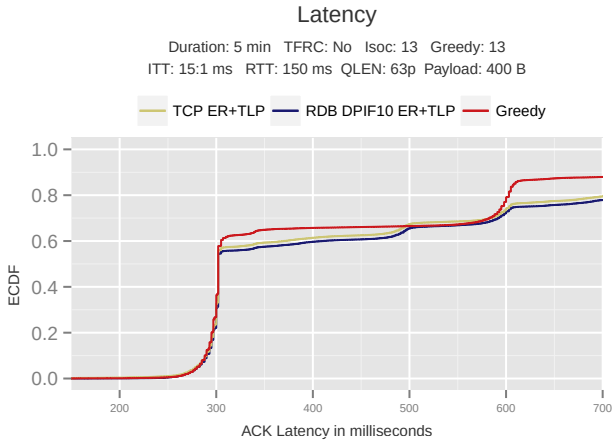
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	90 / 97 / 119	1432/1435/1438	4/1/3/3/4/17
RDB DPIF10 ER+TLP	rdbsender	13	93 / 99 / 105	1439/1443/1444	3/1/3/3/4/17
Greedy	zsender	13	64 / 70 / 76	1439/1445/1447	5/1/4/5/6/20

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
40.7 K	3621	3631	0	58.5 M	53.4 M	5.1 M	5.1 M	8.69 / 8.89	12509
39.2 K	3605	3570	218	56.7 M	51.4 M	5.2 M	5.1 M	9.14 / 9.18	12464
56.2 K	3739	3749	0	81.2 M	75.8 M	5.4 M	5.4 M	6.64 / 6.65	16252

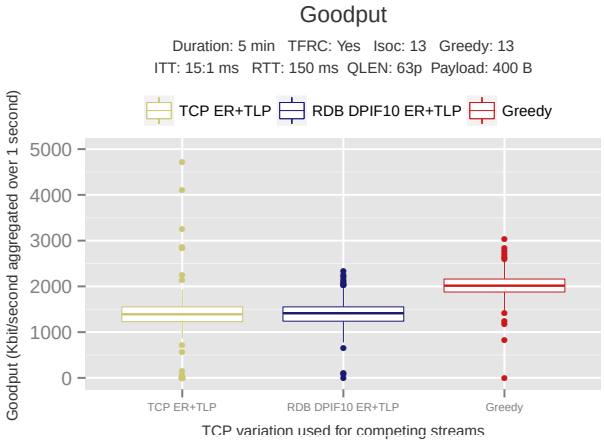
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
489	302	370	584	712	891	908	1098	1147	1189	1199	1335	2781	9820	65550
496	302	411	593	772	896	1007	1106	1167	1189	1199	1211	3213	7742	18100
445	302	305	583	603	850	891	898	901	903	908	1187	3929	9856	11220

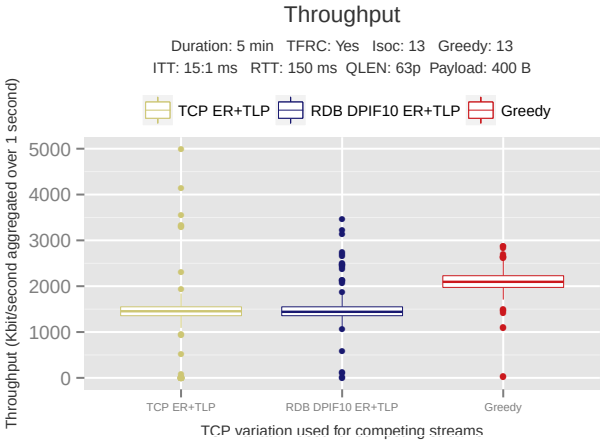
(c)

Figure A.4.41



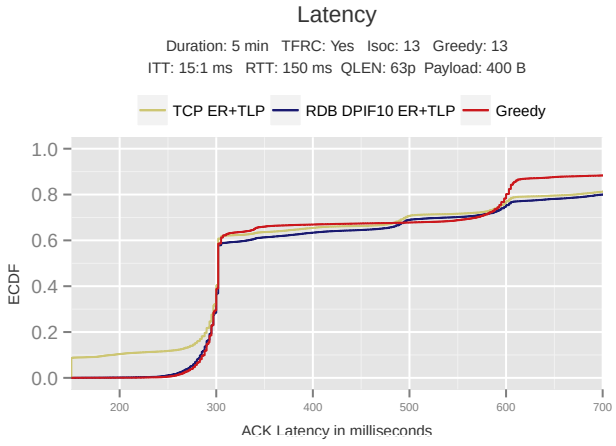
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	86 / 94 / 101	1338/1353/1368	5/1/3/4/5/16
RDB DPIF10 ER+TLP	rdbsender	13	86 / 94 / 102	1401/1431/1443	4/1/3/3/4/57
Greedy	zsender	13	60 / 70 / 79	1445/1446/1447	5/1/4/5/6/15

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
42.4 K	3300	3348	0	57.4 M	52.7 M	4.8 M	4.8 M	8.32 / 7.77	11847
41.3 K	3617	3421	1.7 K	59.1 M	53.0 M	6.1 M	4.9 M	8.71 / 8.75	12178
56.2 K	3650	3664	0	81.3 M	76.0 M	5.3 M	5.3 M	6.49 / 6.49	15977

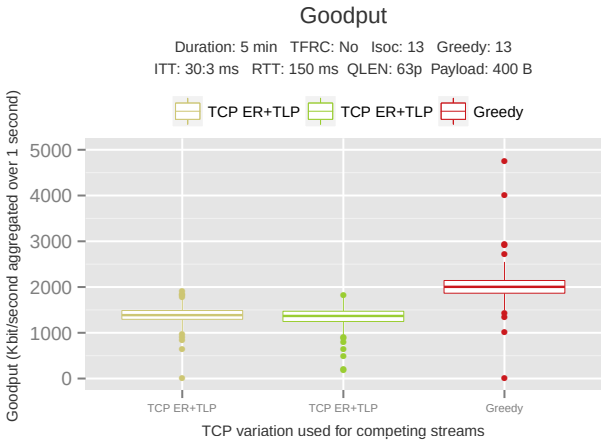
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
452	302	302	496	671	884	902	1080	1111	1181	1196	1206	1998	15314	17670
474	302	333	542	700	889	905	1094	1124	1182	1196	1208	2445	3869	17240
460	302	305	569	600	823	886	896	901	903	910	1191	9862	33481	33760

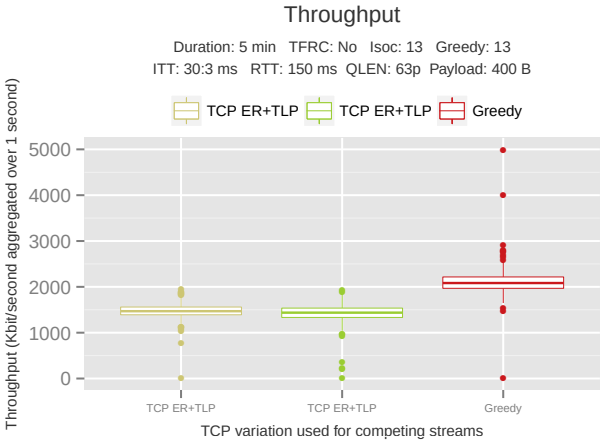
(c)

Figure A.4.42



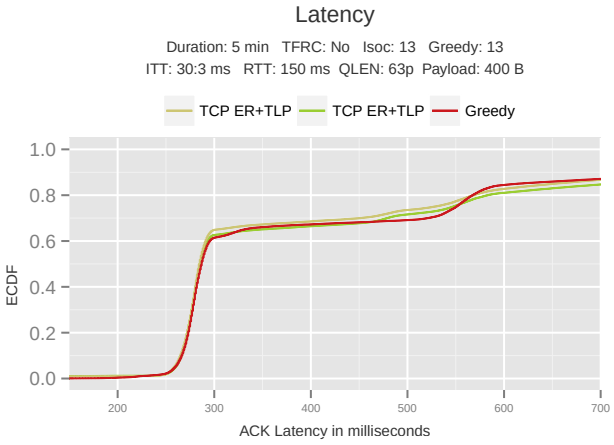
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	72 / 75 / 79	1035/1092/1155	5/1/3/4/6/13
TCP ER+TLP	rdbsender	13	75 / 82 / 89	1079/1171/1265	4/1/3/4/5/13
Greedy	zsender	13	58 / 69 / 74	1444/1446/1447	5/1/4/5/6/26

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
51.5 K	3829	3829	0	56.3 M	52.3 M	3.9 M	3.9 M	6.98 / 7.42	12738
47.1 K	3900	3900	0	55.1 M	50.6 M	4.5 M	4.5 M	8.13 / 8.27	12045
57.0 K	4312	4322	0	82.4 M	76.2 M	6.2 M	6.2 M	7.57 / 7.56	14588

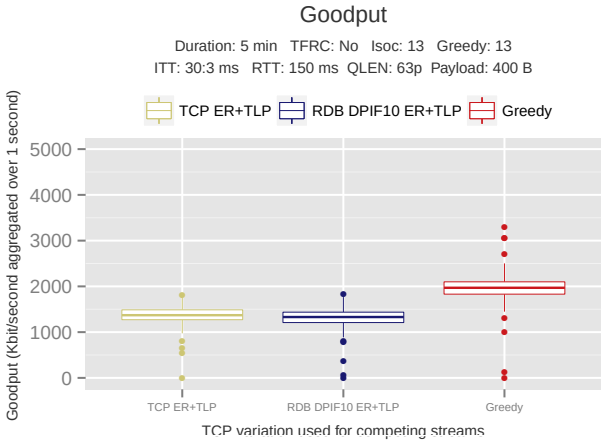
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
414	285	291	451	569	757	819	867	1014	1084	1142	1352	1869	2623	3057
435	285	293	479	583	784	852	1037	1087	1135	1301	1529	2479	3220	8113
435	287	295	521	567	803	835	861	947	1090	1150	1438	5085	11481	13240

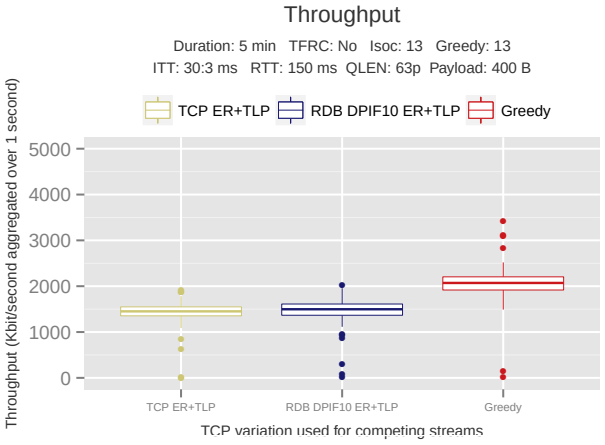
(c)

Figure A.4.43



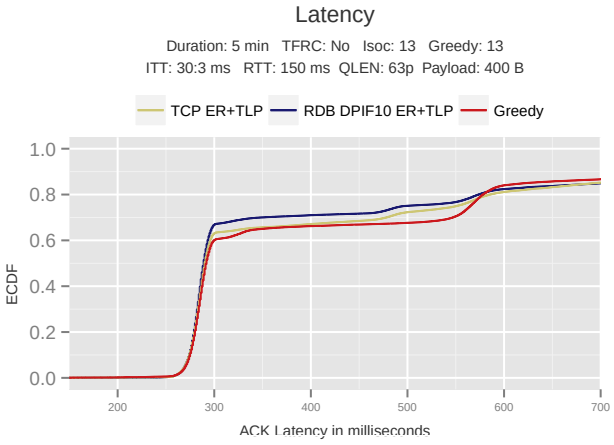
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	73 / 81 / 86	1037/1148/1216	4/1/3/4/6/13
RDB DPIF10 ER+TLP	rdbsender	13	81 / 88 / 91	1271/1322/1356	4/1/3/4/5/13
Greedy	zsender	13	65 / 70 / 80	1443/1445/1447	5/1/3/5/6/22

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
48.6 K	3717	3727	0	55.7 M	51.7 M	4.0 M	4.0 M	7.17 / 7.64	12462
44.0 K	3789	3235	6.5 K	58.1 M	49.5 M	8.6 M	4.3 M	8.56 / 8.61	9458
55.8 K	4233	4243	0	80.7 M	74.5 M	6.1 M	6.1 M	7.59 / 7.58	14757

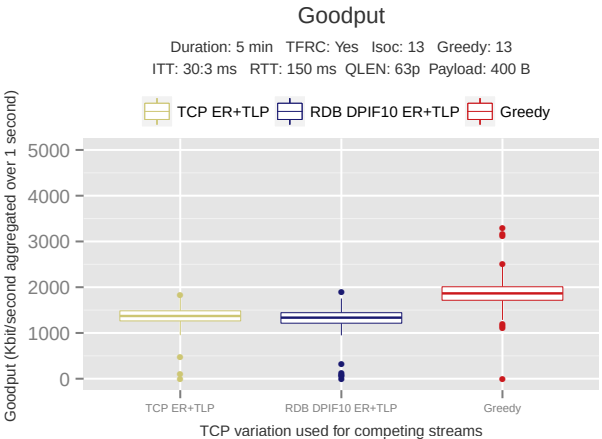
(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
433	290	296	478	586	777	847	912	1059	1121	1168	1399	2080	4271	16230
431	289	293	349	574	807	864	1056	1111	1146	1316	1456	2599	6340	17260
450	292	300	547	578	831	855	874	931	1120	1163	1462	5447	9662	10500

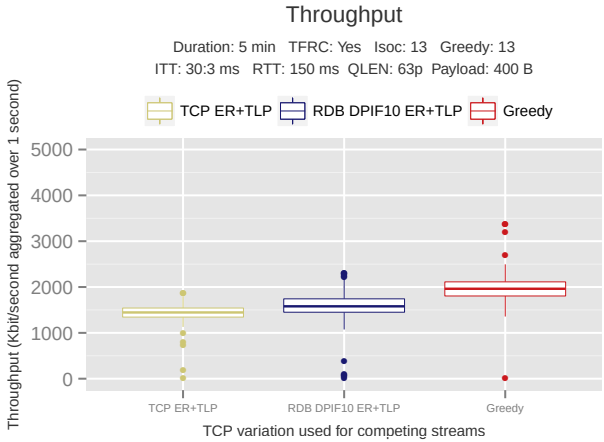
(c)

Figure A.4.44



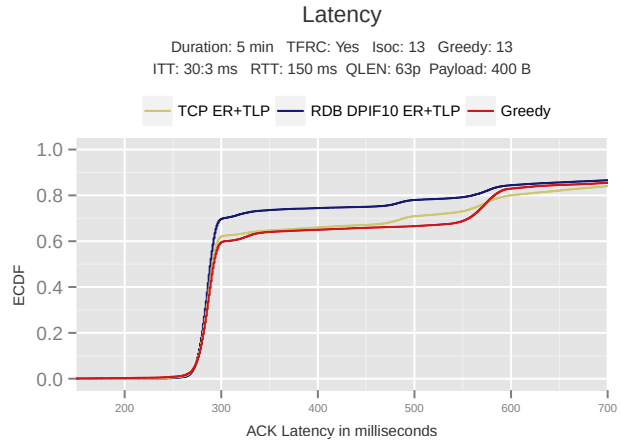
Name	Host	Streams	ITT (avg)	Payload (avg)	CWND
TCP ER+TLP	wsender	13	78 / 83 / 86	1105/1191/1237	4/1/3/4/5/14
RDB DPIF10 ER+TLP	rdbsender	13	73 / 81 / 88	1261/1299/1354	5/1/3/4/6/45
Greedy	zsender	13	67 / 74 / 81	1445/1446/1447	5/1/3/5/6/17

(a)



Sent p	Lost p	Retr p	RDB p	Payl B	Unique B	Rdn B	Retr B	Loss % (B/p)	DupACKs
47.0 K	3835	3845	0	56.0 M	51.6 M	4.4 M	4.4 M	7.77 / 8.15	12202
48.0 K	4205	3128	12.3 K	62.3 M	49.6 M	12.8 M	4.1 M	8.67 / 8.75	9062
53.2 K	4235	4245	0	77.0 M	70.9 M	6.1 M	6.1 M	7.95 / 7.95	14457

(b)



Mean	50th	60th	70th	80th	90th	93th	95th	96th	97th	98th	99th	99.9th	99.99th	Max
445	291	296	489	599	797	861	1038	1089	1138	1190	1438	2666	4828	8162
417	289	292	302	560	781	857	999	1093	1144	1309	1543	2758	3623	4733
460	292	304	556	583	842	862	879	1072	1134	1167	1454	4794	18688	18930

(c)

Figure A.4.45

Appendix B

RDBv2 implementation source code

In this chapter, som key parts of the RDBv2 implementation is listed. The complete source code of RDBv2 can be found at bitbucket.org/bendikro/net-next-rdb.

B.1 Congestion control changes in Linux

Source Code

```
1  /*
2   * TCP Reno congestion control
3   * This is special case used for fallback as well.
4   */
5  /* This is Jacobson's slow start and congestion avoidance.
6   * SIGCOMM '88, p. 328.
7   */
8  void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked)
9  {
10     struct tcp_sock *tp = tcp_sk(sk);
11
12     if (!tcp_is_cwnd_limited(sk))
13         return;
14
15     /* In "safe" area, increase. */
16     if (tp->snd_cwnd <= tp->snd_ssthresh)
17         tcp_slow_start(tp, acked);
18     /* In dangerous area, increase slowly. */
19     else
20         tcp_cong_avoid_ai(tp, tp->snd_cwnd);
21 }
```

This function is located in tcp_cong.c and tests if the stream is application limited

Code Listing B.1: `tcp_is_cwnd_limited` from TCP New Reno that tests if the send rate is limited by the CWND

Source Code

```
/* RFC2861 Check whether we are limited by application or
    congestion window
 * This is the inverse of cwnd check in tcp_tso_should_defer
 * This is the version from tcp_cong.c in kernel v3.15.
 */
bool tcp_is_cwnd_limited(const struct sock *sk, u32 in_flight)
{
    const struct tcp_sock *tp = tcp_sk(sk);
    u32 left;

    if (in_flight >= tp->snd_cwnd)
        return true;

    left = tp->snd_cwnd - in_flight;
    if (sk_can_gso(sk) &&
        left * sysctl_tcp_tso_win_divisor < tp->snd_cwnd &&
        left < tp->xmit_size_goal_segs)
        return true;
    return left <= tcp_max_tso_deferred_mss(tp);
}
```

The implementation of `tcp_is_cwnd_limited` in TCP New Reno (`tcp_cong.c`) from kernel version 3.15

Code Listing B.2: `tcp_is_cwnd_limited` in Linux kernel version 3.15 that tests if the send rate is limited by the CWND

Source Code

```
/* We follow the spirit of RFC2861 to validate cwnd but implement a
    more
 * flexible approach. The RFC suggests cwnd should not be raised
    unless
 * it was fully used previously. And that's exactly what we do in
 * congestion avoidance mode. But in slow start we allow cwnd to
    grow
 * as long as the application has used half the cwnd.
 * Example :
 *    cwnd is 10 (IW10), but application sends 9 frames.
 *    We allow cwnd to reach 18 when all frames are ACKed.
 * This check is safe because it's as aggressive as slow start
    which already
 * risks 100% overshoot. The advantage is that we discourage
    application to
 * either send more filler packets or data to artificially blow up
    the cwnd
 * usage, and allow application-limited process to probe bw more
    aggressively.
 */
static inline bool tcp_is_cwnd_limited(const struct sock *sk)
{
    const struct tcp_sock *tp = tcp_sk(sk);

    /* If in slow start, ensure cwnd grows to twice what was ACKed.
        */
    if (tp->snd_cwnd <= tp->snd_ssthresh)
        return tp->snd_cwnd < 2 * tp->max_packets_out;

```

```
22      return tp->is_cwnd_limited;
23  }
```

This function is located in tcp_cong.c, and tests if the stream is application limited.

Code Listing B.3: tcp_is_cwnd_limited in Linux kernel version 3.16 that tests if the send rate is limited by the CWND

B.2 RDBv2 bundling implementation

Source Code

```
1  static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now,
2                             int nonagle, int push_one, gfp_t gfp)
3  {
4    while ((skb = tcp_send_head(sk))) {
5         ....
6      /* Does at least the first segment of SKB fit into the send
           window? */
7      if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now)))
8           break;
9         ....
10        TCP_SKB_CB(skb)->when = tcp_time_stamp;
11
12        // Code added to support RDB
13        if (unlikely(inet_csk(sk)->icsk_ca_ops->pkt_send)) {
14            loop_counter++;
15            /* If 1, SKB has been sent by the function */
16            if (likely(inet_csk(sk)->icsk_ca_ops->
17                pkt_send(skb, mss_now, sk, tp,
18                         gfp, loop_counter, false))) {
19                goto repair;
20            }
21        }
22        // End: code added to support RDB
23
24        if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
25            break;
26 repair:
27        /* Advance the send_head.  This one is sent out.
28         * This call will increment packets_out.
29         */
30        tcp_event_new_data_sent(sk, skb);
31        ....
32 }
```

Code from tcp_write_xmit in tcp_output.c. Removed code lines that are not directly relevant is indicated by "...."

Code Listing B.4: Excerpt from the function tcp_write_xmit in tcp_output.c

Source Code

```
1  /*
2   * do_rdb() - Try to create and send an RDB packet
3   * xmit_skb: The original SKB to be sent
4   * loop_count: The loop iteration index
5   * retrans: If this is a retransmit
6   *
7   * Return: true if successfully sent RDB packet, else false
8   */
9  bool do_rdb(struct sk_buff *xmit_skb, unsigned int mss_now, struct
       sock *sk, struct tcp_sock *tp, gfp_t gfp_mask, int loop_count,
       bool retrans) {
10     struct sk_buff *rdb_skb = NULL;
11     struct inet_connection_sock *icsk;
12     struct RDBcong *cong = inet_csk_ca(sk);
13     int ret = false;
14
15     // No RDB on retrans, for now
16     if (retrans)
17         goto end;
18
19     // How we detect that RDB was used on this SKB
20     RDB_SKB_CB(xmit_skb)->rdb_start_seq = 0;
21
22     if (run_rdb(cong, tp)) {
23         rdb_skb = redundant_data_bundling(xmit_skb, mss_now, sk,
               gfp_mask, retrans);
24         if (!rdb_skb)
25             goto end;
26         icsk = inet_csk(sk);
27         skb_mstamp_get(\&xmit_skb->skb_mstamp);
28         rdb_skb->tstamp.tv64 = 0;
29         ret = !tcp_transmit_skb(sk, rdb_skb, 0, gfp_mask);
30     }
31  end:;
32     if (use_tfrc) {
33         // Use the original SKB
34         if (ret) {
35             // We use the RDB SKB
36             rdb_tx_update_s(cong, xmit_skb->len, rdb_skb->len);
37         } else {
38             rdb_tx_update_s(cong, xmit_skb->len, xmit_skb->len);
39         }
40         cong->tfrc.tx_stats->tx_t_last_sent = ktime_get_real();
41     }
42     return ret;
43 }
```

The function `do_rdb` implemented in `rdb_cc.c`. The code has been stripped down to only the most important parts

Code Listing B.5

Source Code

```
1 inline bool run_rdb(struct RDBcong *cong, struct tcp_sock *tp) {
2     return (tp->thin_rdb) &&
3         (cong->dynamic_pif ? tcp_stream_is_thin_dpif(tp, cong->
              minrtt_us) : tcp_stream_is_thin_spif(tp));
4 }
```

The function run_rdb implemented in rdb_cc.c.

Code Listing B.6

Source Code

```
1  /**
2   * tcp_stream_is_thin_dpif() - Tests if the stream is thin based in
        dynamic PIF limit
3   * min_rtt_us: The minimun RTT registered for the connection
4   *
5   * Return: true if current PIF count is lower than the dynamic PIF
       limit, else false
6   */
7  inline bool tcp_stream_is_thin_dpif(struct tcp_sock *tp, u32
       min_rtt_us) {
8      // Div by 1000 to get ms. Div by dynamic_pif, the
9      // minimum allowed ITT (Inter-transmission time) in ms
10     return tp->packets_out < ((min_rtt_us >> 3) / 1000 /
           dynamic_pif);
11 }
```

The function tcp_stream_is_thin_dpif implemented in rdb_cc.c.

Code Listing B.7: The new function in RDBv2 for classifying thin streams based on a DPIFL.

Source Code

```
1 static unsigned int tcp_stream_is_thin_spif(struct tcp_sock *tp)
2 {
3     return tp->packets_out < sysctl_tcp_thin_packet_limit;
4 }
```

The function tcp_stream_is_thin_spif implemented in rdb_cc.c.

Code Listing B.8: A modification to tcp_stream_is_thin used in the experiments with different SPIFL values.

Source Code

```
1  /*
2   * redundant_data_bundling() - Tries to perform RDB
3   * xmit_skb: The SKB that should be sent
4   * retrans: If this is a retransmit
5   *
6   * Return: A new SKB containing redundant data, or NULL if no
        bundling could be performed
7   */
8  struct sk_buff* redundant_data_bundling(struct sk_buff *xmit_skb,
       unsigned int mss_now, struct sock *sk, gfp_t gfp_mask, bool
       retrans) {
9      u32 byte_count_in_rdb_skb;
10     u32 skb_data_offset;
11     u32 skb_in_bundle_count;
12     struct sk_buff* first_to_bundle, *rdb_skb;
13     ....
14     // First in write queue, means no previous SKB to merge with
15     if (!retrans && skb_queue_is_first(&sk->sk_write_queue,
           xmit_skb)) {
16         return NULL;
17     }
18
19     // Previous has been SACKED
20     if (TCP_SKB_CB(xmit_skb->prev)->sacked & TCPCB_SACKED_ACKED) {
21         return NULL;
22     }
23
24     // Find number of (previous) SKBs to get data from
25     first_to_bundle = rdb_can_bundle_check(sk, xmit_skb, mss_now,
26                                            &byte_count_in_rdb_skb,
27                                            &skb_data_offset,
28                                            &skb_in_bundle_count);
29     if (!first_to_bundle) {
30         return NULL;
31     }
32
33     // Create an SKB that contains the data from '
           skb_in_bundle_count' SKBs.
34     rdb_skb = create_rdb_skb(first_to_bundle, xmit_skb, sk,
35                              gfp_mask, byte_count_in_rdb_skb,
36                              skb_data_offset);
37     if (!rdb_skb) {
38         return NULL;
39     }
40     ....
41     return rdb_skb;
42 }
```

Code from `tcp_write_xmit` in `tcp_output.c`. Removed code lines that are not directly relevant is indicated by "...."

Code Listing B.9

Source Code

```
1  /*
2   * rdb_can_bundle_check() - check if redundant data can be bundled
3   * skb: The SKB with the newest data to be bundled
4   * byte_count_in_rdb_skb: Will contain the resulting number
5   *                        of bytes to bundle at exit.
6   * skb_data_offset: The data offset in the first SKB that will
7   *                  be in the bundle
8   * skbs_to_bundle_count: The total number of SKBs to be in the
9   *                       bundle
10  *
11  * Traverses the entire write qeueue and checks if there is
12  * data to be bundled.
13  *
14  * Return: The first SKB to be in the bundle, or NULL if
15  * no bundling.
16  */
17 static struct sk_buff* rdb_can_bundle_check(struct sock *sk, struct
        sk_buff *xmit_skb, unsigned int mss_now, u32 *
       byte_count_in_rdb_skb, u32 *skb_data_offset, u32 *
       skbs_to_bundle_count) {
18     struct sk_buff *first_to_bundle = NULL;
19     struct sk_buff *tmp, *skb = xmit_skb->prev;
20     u32 skbs_in_bundle_count = 1;
21     u32 offset_in_skb = 0;
22     u32 byte_count = xmit_skb->len;
23     u32 tp_snd_una = tcp_sk(sk)->snd_una;
24
25     if (!mss_now)
26         mss_now = tcp_sk(sk)->mss_cache;
27
28     if (skb_queue_is_first(&sk->sk_write_queue, xmit_skb))
29         return NULL;
30
31     // We start at the first skb, and go backwards in the list.
32     tcp_for_write_queue_reverse_from_safe(skb, tmp, sk) {
33
34         // This SKB does not contain unacked data, so break off
               here.
35         if (TCP_SKB_CB(skb)->end_seq < tp_snd_una)
36             break;
37
38         // Not enough room to bundle data from this SKB
39         if ((byte_count + skb->len) > mss_now) {
40             break;
41         }
42
43         // Exceeds max_bundle_bytes
44         if (sysctl_tcp_rdb_max_bundle_bytes && ((byte_count + skb->
               len) > sysctl_tcp_rdb_max_bundle_bytes)) {
45             break;
46         }
47
48         // Exceeds max_bundle_skbs
49         if (sysctl_tcp_rdb_max_bundle_skbs && (skbs_in_bundle_count
                > sysctl_tcp_rdb_max_bundle_skbs)) {
50             break;
51         }
52
53         byte_count += skb->len;
54         // Calculate the offset in this SKB. If tp_snd_una is
               bigger than seq,
55         // the first byte in the rdb-bundled SKB will be tp_snd_una
               .
```

```
56          offset_in_skb = max(TCP_SKB_CB(skb)->seq, tp_snd_una) -
                TCP_SKB_CB(skb)->seq;
57          skbs_in_bundle_count++;
58          first_to_bundle = skb;
59      }
60
61      *skb_data_offset = offset_in_skb;
62      *byte_count_in_rdb_skb = byte_count;
63      *skbs_to_bundle_count = skbs_in_bundle_count;
64      return first_to_bundle;
65  }
```

The function rdb_can_bundle_check in rdb_skb.c.

Code Listing B.10

B.3 TFRC simulations

TFRC-SP simulation: Throughput

```
1  def calc_throughput(hist, tp_packet_size_func, hdr_accounting=False
       ):
2    """
3    X is the transmit rate in bytes/second.
4    s is the packet size in bytes.
5    R is the round trip time in seconds.
6    p is the loss event rate, between 0 and 1.0, of the
7      number of loss events as a fraction of the number
8      of packets transmitted.
9    t_RTO is the TCP retransmission timeout value in seconds.
10   b is the number of packets acknowledged by a single TCP
11     acknowledgement.
12                                s
13   X =  ---------------------------------------------------
14      R*sqrt(2*b*p/3) + (t_RTO * (3*sqrt(3*b*p/8) * p * (1+32*p^2)))
15   """
16   x = (hist.rtt * sqrt(2 * hist.b * hist.p / 3.0) +
17        (hist.t_rto * (3 * sqrt(3 * hist.b * hist.p / 8.0)
18                       * hist.p * (1 + 32 * pow(hist.p, 2)))))
19   x = tp_packet_size_func(hist) / x
20   packets = x / hist.avg_size
21   new_x = (x * hist.avg_size) / (hist.avg_size + 40)
22   return new_x if hdr_accounting else x
```

Code Listing B.11: Function that calculates the send rate according to equation 2.2

TFRC-SP simulation: Loss Event Rate

```
def loss_event_rate_sp(hist):
   I_tot0 = 0
   I_tot1 = 0
   W_tot = 0
   k_tot0 = 0
   k_tot1 = 0
   for i in range(len(hist.intervals)):
      I_i = hist.intervals[i].length
      if i > 0:
         interval_len = hist.intervals[i].t_start - hist.intervals[
             i - 1].t_start
         rtt_tdelta = datetime.timedelta(microseconds=(self.rtt *
             1000000))
         # This interval is short
         if interval_len < rtt_tdelta:
            I_i = hist.intervals[i].length / hist.intervals[i].lost
      I_tot0 += (I_i * tfrc_weights[i])
      W_tot += tfrc_weights[i]
      if i > 0:
         I_tot1 += (I_i * tfrc_weights[i-1])

   I_tot = max(I_tot0, I_tot1)
   try:
      I_mean = I_tot/float(W_tot)
      p = 1 / I_mean
   except ZeroDivisionError, e:
      raise
   return p
```

Code Listing B.12: Function that calculates the TFRC-SP loss event rate based on pseudo code 4.3.

B.4 TFRC Congestion Control implementation

Source Code

```
struct tcp_congestion_ops {
   struct list_head list;
   unsigned long flags;

   /* initialize private data (optional) */
   void (*init)(struct sock *sk);
   /* cleanup private data  (optional) */
   void (*release)(struct sock *sk);

   /* return slow start threshold (required) */
   u32 (*ssthresh)(struct sock *sk);
   /* lower bound for congestion window (optional) */
   u32 (*min_cwnd)(const struct sock *sk);
   /* do new cwnd calculation (required) */
   void (*cong_avoid)(struct sock *sk, u32 ack, u32 in_flight);
   /* call before changing ca_state (optional) */
   void (*set_state)(struct sock *sk, u8 new_state);
   /* call when cwnd event occurs (optional) */
   void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
   /* new value of cwnd after loss (optional) */
```

```
21     u32  (*undo_cwnd)(struct sock *sk);
22     /* hook for packet ack accounting (optional) */
23     void (*pkts_acked)(struct sock *sk, u32 num_acked, s32 rtt_us);
24     /* call to send packet from tcp_write_xmit (optional) */
25     bool (*pkt_send)(struct sk_buff *xmit_skb, unsigned int mss_now,
26                      struct sock *sk, struct tcp_sock *tp,
27                      gfp_t gfp_mask, int loop_count,
28                      bool retrans);
29     /* hook for ack received that removes packets from the write
           queue (optional) */
30     void (*tcp_ack)(struct sock *sk, struct sk_buff *skb, int flag,
31                     u32 prior_fackets, u32 prior_snd_una,
32                     s32 sack_rtt);
33     /* get info for inet_diag (optional) */
34     void (*get_info)(struct sock *sk, u32 ext, struct sk_buff *skb);
35
36     char name[TCP_CA_NAME_MAX];
37     struct module   *owner;
38 };
```

Defined in `include/net/tcp.h`

Code Listing B.13: Entry points defined in `struct tcp_congestion_ops` for the Linux CC framework in RDBv2

Source Code

```
1  /* This routine deals with incoming acks, but not outgoing ones. */
2  static int tcp_ack(struct sock *sk, const struct sk_buff *skb, int
       flag) {
3      ....
4      u32 prior_snd_una = tp->snd_una;
5      u32 ack_seq = TCP_SKB_CB(skb)->seq;
6      u32 ack = TCP_SKB_CB(skb)->ack_seq;
7      int acked = 0; /* Number of packets newly acked */
8      ....
9      if (!(flag & FLAG_SLOWPATH) && after(ack, prior_snd_una)) {
10         ....
11         tcp_ca_event(sk, CA_EVENT_FAST_ACK);
12         ....
13     } else {
14         ...
15         tcp_ca_event(sk, CA_EVENT_SLOW_ACK);
16     }
17     ....
18     /* See if we can take anything off of the retransmit queue. */
19     acked = tp->packets_out;
20     flag |= tcp_clean_rtx_queue(sk, prior_fackets, prior_snd_una,
           sack_rtt_us);
21     acked -= tp->packets_out;
22
23     /* Advance cwnd if state allows */
24     if (tcp_may_raise_cwnd(sk, flag))
25         tcp_cong_avoid(sk, ack, acked);
26     ....
27 }
```

The function `tcp_ack` located in `tcp_input.c` handles incomming ACKs.

Code Listing B.14: `tcp_ack` function in the TCP engine handling incoming ACKs

Source Code

```
1  /*
2   * rdb_ack() - Perform loss detection
3   */
4  void rdb_ack(struct sock *sk) {
5      struct tcp_sock *tp = tcp_sk(sk);
6      struct RDBcong *cong = inet_csk_ca(sk);
7      struct sk_buff *acked = NULL;
8      struct sk_buff *lost = NULL;
9      int lost_count;
10
11     lost_count = rdb_check_rtx_queue_acked(sk, tp->snd_una, &acked,
           &lost);
12     if (lost_count) {
13         cong->rdb_ack_loss_count += lost_count;
14         cong->rdb_ack_loss_count_total += lost_count;
15     }
16
17     if (use_tfrc) {
18         cong->tfrc.tx_stats->tx_rtt = tp->srtt_us >> 3;
19         rdb_tfrc_update_loss_history(cong, sk, lost_count, lost);
20     }
21 }
```

The function rdb_ack located in rdb_cc.c performs loss detection from incomming ACKs.

Code Listing B.15: Function rdb_ack in RDBv2

Source Code

```
1  /**
2   * rdb_check_rtx_queue_acked() - Perform loss detection
3   *                              by analysing acks.
4   * sk: the socket.
5   * seq_acked: The sequence number that was acked.
6   * acked: The sk_buff which will be set to the SKB
7   *        (with highest seqnum) that was acked.
8   * lost: The sk_buff which will be set to the first SKB
9   *       (with lowest seqnum) that was lost.
10  *
11  * Return: The number of packets that are presumed to be lost.
12  */
13 static int rdb_check_rtx_queue_acked(struct sock *sk, u32 seq_acked
        , struct sk_buff **acked, struct sk_buff **lost) {
14     struct tcp_sock *tp = tcp_sk(sk);
15     struct sk_buff *skb, *tmp, *prev_skb = NULL;
16     struct sk_buff *send_head = tcp_send_head(sk);
17     struct tcp_skb_cb *scb;
18     int fully_acked = true;
19     int lost_count = 0;
20
21     tcp_for_write_queue(skb, sk) {
22         if (skb == send_head)
23             break;
24
25         scb = TCP_SKB_CB(skb);
26
```

```
27          /* Determine how many packets and what bytes were acked, no
                TSO support */
28          if (after(scb->end_seq, tp->snd_una)) {
29              if (tcp_skb_pcount(skb) == 1 ||
30                  !after(tp->snd_una, scb->seq)) {
31                  break;
32              }
33
34              // We do not handle SKBs with gso_segs
35              if (tcp_skb_pcount(skb)) {
36                  break;
37              }
38              fully_acked = false;
39          }
40
41          // Acks up to this packet
42          if (scb->end_seq == seq_acked) {
43              *acked = skb;
44              // This was sent with RDB data, and this skb acked data
                    on previous skb
45              if (RDB_SKB_CB(skb)->rdb_start_seq != 0 && prev_skb) {
46                  tcp_for_write_queue(tmp, sk) {
47                      if (tmp == skb) // We have reached the acked
                            SKB
48                          break;
49                      lost_count++;
50                      if (*lost == NULL)
51                          *lost = skb;
52                  }
53              }
54              break;
55          }
56          if (!fully_acked)
57              break;
58          prev_skb = skb;
59      }
60      return lost_count;
61  }
```

The function `rdb_check_rtx_queue_acked` located in `rdb_cc.c` traverses the TCP output queue to detect lost packet.

Code Listing B.16: Function `rdb_check_rtx_queue_acked` in RDBv2

Source Code

```
1  %
2  /**
3   * rdb_tfrc_update_loss_history() - Update the loss history.
4   * cong: The RDBCong struct for the connection.
5   * lost_count: The number of packets believed to be lost.
6   * lost: The first SKB believed to be lost.
7   */
8  void rdb_tfrc_update_loss_history(struct RDBcong *cong, struct sock
       *sk, int lost_count, struct sk_buff *lost) {
9      struct rdb_tfrc_tx_stats* tx_stats = cong->tfrc.tx_stats;
10
11     if (lost_count) {
12         rdb_tfrc_handle_loss(cong->tfrc.li_hist, lost, lost_count,
               tx_stats->tx_rtt, rdb_tfrc_first_li, sk);
```

```
13          tfrc_lh_calc_i_mean_sp(cong->tfrc.li_hist, tx_stats->tx_rtt
                );
14      }
15      else {
16          // Increase interval length
17          rdb_tfrc_lh_update_i_mean(&cong->tfrc, 0);
18      }
19
20      if (cong->tfrc.li_hist->i_mean) {
21          tx_stats->tx_p = scaled_div(1, cong->tfrc.li_hist->i_mean);
22      }
23
24      /* Update sending rate (step 4 of [RFC 3448, 4.3]) */
25      if (tx_stats->tx_p > 0) {
26         // Calculate the send rate to use
27          u32 tx_x_calc = tfrc_calc_x(tx_stats->tx_s,
28                                      tx_stats->tx_rtt,
29                                      tx_stats->tx_p);
30          // We adjust the send rate by accounting for the network
31          // and transport layer headers
32          tx_x_calc = (tx_x_calc * tx_stats->tx_s) / (tx_stats->tx_s
                + 40);
33          tx_stats->tx_x_calc = tx_x_calc;
34      }
35      // Recalculate allowed sending rate X
36      rdb_tx_update_x(cong, sk, NULL);
37  }
```

The function rdb_tfrc_update_loss_history located in rdb_tfrc.c.

Code Listing B.17: Function rdb_tfrc_update_loss_history in RDBv2

Source Code

```
1  %
2  /**
3   * rdb_tx_update_x() - Update allowed sending rate X
4   * cong: The RDBCong struct for the connection
5   * sk: the socket
6   * stamp: most recent time if available (can be left NULL).
7   *
8   * This function borrows from ccid3_hc_tx_update_x in dccp/ccids/
        ccid3.c (12/2014)
9   */
10 void rdb_tx_update_x(struct RDBcong *cong, struct sock *sk, ktime_t
        *stamp) {
11      struct rdb_tfrc_tx_stats* tx_stats = cong->tfrc.tx_stats;
12
13      if (!tx_stats->tx_rtt)
14          return;
15
16      __u64 min_rate = tx_stats->tx_x >> 1; // Minimum rate is half
            the current rate
17
18      const __u64 old_x = tx_stats->tx_x;
19      ktime_t now = stamp ? *stamp : ktime_get_real();
20      u32 delta = ktime_us_delta(now, tx_stats->tx_t_last_sent);
21      u32 idle_rtts = delta / tx_stats->tx_rtt;
22
23      /*
```

```
24      * Handle IDLE periods: do not reduce below RFC3390 initial
            sending rate
25      * when idling [RFC 4342, 5.1]. Definition of idling is from
            rfc3448bis:
26      * a sender is idle if it has not sent anything over a 2-RTT-
            period.
27      * For consistency with X and X_recv, min_rate is also scaled
            by 2^6.
28      */
29     if (idle_rtts >= 2) {
30         min_rate = rfc3390_initial_rate(tx_stats);
31     }
32
33     if (tx_stats->tx_p > 0) {
34         tx_stats->tx_x = max(((__u64)tx_stats->tx_x_calc) << 6,
               min_rate);
35     } else if (ktime_us_delta(now, tx_stats->tx_t_ld) - (s64)
           tx_stats->tx_rtt >= 0) {
36         tx_stats->tx_x = max(tx_stats->tx_x,
37                              scaled_div(((__u64)tx_stats->tx_s) <<
                                     6, tx_stats->tx_rtt));
38         tx_stats->tx_t_ld = now;
39     }
40
41     if (tx_stats->tx_x != old_x) {
42         u64 rtts_per_second = div64_u64(1000000, tx_stats->tx_rtt);
43         // Update the CWND value based on the calculated throughput
44         if (rtts_per_second) {
45             tx_stats->tx_x_rtt = div64_u64(tx_stats->tx_x >> 6,
                   rtts_per_second);
46             tx_stats->tx_x_cwnd = div64_u64((__u64) tx_stats->
                   tx_x_rtt, tx_stats->tx_s);
47         }
48     }
49 }
```

The function `rdb_tx_update_x` located in `rdb_tfrc.c`.

Code Listing B.18: Function `rdb_tx_update_x` in RDBv2

Source Code

```
1  %
2  /**
3   * tfrc_lh_calc_i_mean_sp() - Calculate the loss event rate
        according to TFRC-SP (Small-Packet variant)
4   * lh:  Loss Interval history
5   * rtt: Current rtt
6   *
7   * Note: Based on tfrc_lh_calc_i_mean from dccp/ccids/lib/
        loss_interval.c
8   */
9  void tfrc_lh_calc_i_mean_sp(struct tfrc_loss_hist *lh, u32 rtt) {
10     u32 i_i, i_tot0 = 0, i_tot1 = 0, w_tot = 0;
11     int i, k = tfrc_lh_length(lh) - 1; /* k is as in rfc3448bis,
           5.4 */
12     struct rdb_tfrc_loss_interval *tmp, *prev;
13     u32 i_duration;
14     bool cur_i_short = false;
15     struct skb_mstamp now;
16
```

```
17      if (k <= 0)
18          return;
19
20      skb_mstamp_get(&now);
21
22      // We start at the newest intervals (index 0)
23      for (i = 0; i <= k; i++) {
24          tmp = tfrc_lh_get_interval(lh, i);
25          i_i = tfrc_lh_get_interval_len(lh, i);
26
27          if (i > 0) {
28              // It's a short loss interval that spans less than two
                    RTTs
29              if (tmp->is_short) {
30                  i_i = div64_u64(tmp->li_length, tmp->li_lost);
31              }
32              i_tot1 += i_i * tfrc_lh_weights[i-1];
33          }
34          else {
35              i_duration = skb_mstamp_us_delta(&now, &tmp->li_mstamp)
                    ;
36              cur_i_short = i_duration < (rtt * 2);
37          }
38
39          if (i < k) {
40              i_tot0 += i_i * tfrc_lh_weights[i];
41              w_tot += tfrc_lh_weights[i];
42          }
43          prev = tmp;
44      }
45
46      if (cur_i_short) {
47          lh->i_mean = i_tot1;
48      }
49      else {
50          lh->i_mean = max(i_tot0, i_tot1);
51      }
52      /* Handle the extra restriction defined in rfc4828 to only
            include the latest interval if it's older than 2 * RTT. */
53      if (k > 1) {
54          struct rdb_tfrc_loss_interval *cur = tfrc_lh_peek(lh);
55          ktime_t now = ktime_get_real();
56          s64 diff = ktime_us_delta(now, cur->li_tstamp);
57
58          //  If the current loss interval I_0 is "short"
59          if (diff < (rtt * 2)) {
60              lh->i_mean = i_tot1;
61          }
62      }
63      lh->i_mean /= w_tot;
64  }
```

The function `tfrc_lh_calc_i_mean_sp` located in `loss_history.c` that calculates the average loss interval.

Code Listing B.19: Function `tfrc_lh_calc_i_mean_sp` in RDBv2

Source Code

```
/**
 * tcp_rdbcong_avoid_tfrc() - Perform congstion avoidance
 *                           with TFRC CC
 * sk: the socket
 * ack: The sequence number that was recently acked
 * acked: The number of packets newly acked
 */
void tcp_rdbcong_avoid_tfrc(struct sock *sk, u32 ack, u32 acked) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct RDBcong *cong = inet_csk_ca(sk);

    /*  Update lowest rtt */
    if (tp->srtt_us < cong->minrtt_us) {
        cong->minrtt_us = tp->srtt_us;
    }

    // Packets were registered lost
    if (cong->rdb_ack_loss_count) {
        tp->snd_ssthresh = inet_csk(sk)->icsk_ca_ops->ssthresh(sk);
        tp->snd_cwnd = cong->tfrc.tx_stats->tx_x_cwnd;
        tp->snd_cwnd_cnt = 0;
        tp->snd_cwnd_stamp = tcp_time_stamp;
        cong->rdb_ack_loss_count = 0;
    }
    else {
        // Do reno
        tcp_reno_cong_avoid_3_15(sk, ack, acked);
    }
    return;
}
```

The function `tcp_rdbcong_avoid_tfrc` located in `rdb_cc.c`. This function is used when the TFRC loss response mechanism is enabled.

Code Listing B.20: Function `tcp_rdbcong_avoid_tfrc` located in `rdb_cc.c`.

Source Code

```
/**
 * tcp_rdbcong_avoid() - Perform congstion avoidance
 * sk: the socket
 * ack: The sequence number that was recently acked
 * acked: The number of packets newly acked
 */
void tcp_rdbcong_avoid(struct sock *sk, u32 ack, u32 acked)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct RDBcong *cong = inet_csk_ca(sk);

    /*  Update lowest rtt */
    if (tp->srtt_us < cong->minrtt_us) {
        cong->minrtt_us = tp->srtt_us;
    }

    // Packets were registered lost
    if (cong->rdb_ack_loss_count) {
        cong->rdb_ack_loss_count = 0;
        tcp_enter_cwr(sk, true);
```

```
21      }
22      else {
23          // Do reno
24          tcp_reno_cong_avoid_3_15(sk, ack, acked);
25      }
26  }
```

The function `tcp_rdbcong_avoid` located in `rdb_cc.c`. This function is used when the TFRC loss response mechanism is enabled

Code Listing B.21: Function `tcp_rdbcong_avoid` located in `rdb_cc.c`

Source Code

```
1 /* Slow start threshold is half the congestion window (min 2) */
2 u32 tcp_reno_ssthresh(struct sock *sk) {
3     const struct tcp_sock *tp = tcp_sk(sk);
4     return max(tp->snd_cwnd >> 1U, 2U);
5 }
```

The function `tcp_reno_ssthresh` located in `tcp_cong.c` returns the slow-start threshold.

Code Listing B.22: `ssthresh` implementation for TCP New Reno

Source Code

```
1 /* Slow start threshold is set to a quarter of the congestion
      window (min 2) */
2 u32 tcp_rdb_ssthresh(struct sock *sk) {
3     const struct tcp_sock *tp = tcp_sk(sk);
4     return max(tp->snd_cwnd >> 2U, 2U);
5 }
```

The function `tcp_rdb_ssthresh` located in `rdb_cc.c` in the RDBv2 returns the slow-start threshold.

Code Listing B.23: `ssthresh` implementation in RDBv2

Appendix C

Patches

C.1 RDB prototype v1 patch

Source Code

```
From ff5c5eb79947186625a0d9d62c5c2bc9cb666840 Mon Sep 17 00:00:00 2001
From: Andreas Petlund <apetlund@simula.no>
Date: Tue, 17 Mar 2009 12:14:35 +0100
Subject: [PATCH] Fixed exp. b.o. SYN bug

---
 include/linux/sysctl.h     |    5 +
 include/linux/tcp.h        |    8 ++
 include/net/sock.h         |    5 +-
 include/net/tcp.h          |   20 +++++
 net/ipv4/sysctl_net_ipv4.c |   32 +++++++
 net/ipv4/tcp.c             |  195 ++++++++++++++++++++++++++++++++++++++++++-
 net/ipv4/tcp_input.c       |  148 ++++++++++++++++++++++++++++++++-
 net/ipv4/tcp_output.c      |  181 ++++++++++++++++++++++++++++++++++++++++
 net/ipv4/tcp_timer.c       |   24 +++++-
 9 files changed, 592 insertions(+), 26 deletions(-)

diff --git a/include/linux/sysctl.h b/include/linux/sysctl.h
index 483050c..f0edacd 100644
--- a/include/linux/sysctl.h
+++ b/include/linux/sysctl.h
@@ -355,6 +355,11 @@ enum
   NET_IPV4_ROUTE=18,
   NET_IPV4_FIB_HASH=19,
   NET_IPV4_NETFILTER=20,
+
+ NET_IPV4_TCP_FORCE_THIN_RDB=29,          /* Added @ Simula */
+ NET_IPV4_TCP_FORCE_THIN_RM_EXPB=30,      /* Added @ Simula */
+ NET_IPV4_TCP_FORCE_THIN_DUPACK=31,       /* Added @ Simula */
+ NET_IPV4_TCP_RDB_MAX_BUNDLE_BYTES=32,    /* Added @ Simula */

   NET_IPV4_TCP_TIMESTAMPS=33,
   NET_IPV4_TCP_WINDOW_SCALING=34,
diff --git a/include/linux/tcp.h b/include/linux/tcp.h
index c6b9f92..c11a564 100644
--- a/include/linux/tcp.h
```

```
37 +++ b/include/linux/tcp.h
38 @@ -97,6 +97,10 @@ enum {
39  #define TCP_CONGESTION   13  /* Congestion control algorithm
       */
40  #define TCP_MD5SIG   14  /* TCP MD5 Signature (RFC2385) */
41
42 +#define TCP_THIN_RDB            15      /* Added @ Simula -
       Enable redundant data bundling  */
43 +#define TCP_THIN_RM_EXPB        16      /* Added @ Simula -
       Remove exponential backoff  */
44 +#define TCP_THIN_DUPACK         17      /* Added @ Simula -
       Reduce number of dupAcks needed */
45 +
46  #define TCPI_OPT_TIMESTAMPS  1
47  #define TCPI_OPT_SACK    2
48  #define TCPI_OPT_WSCALE    4
49 @@ -296,6 +300,10 @@ struct tcp_sock {
50    u8  nonagle;  /* Disable Nagle algorithm?             */
51    u8  keepalive_probes; /* num of allowed keep alive probes */
52
53 + u8      thin_rdb;       /* Enable RDB
                                */
54 + u8      thin_rm_expb;   /* Remove exp. backoff
                           */
55 + u8      thin_dupack;    /* Remove dupack
                            */
56 +
57  /* RTT measurement */
58    u32 srtt;   /* smoothed round trip time << 3  */
59    u32 mdev;   /* medium deviation     */
60 diff --git a/include/net/sock.h b/include/net/sock.h
61 index dfeb8b1..af831d1 100644
62 --- a/include/net/sock.h
63 +++ b/include/net/sock.h
64 @@ -462,7 +462,10 @@ static inline void sk_stream_set_owner_r(
       struct sk_buff *skb, struct sock *sk)
65
66  static inline void sk_stream_free_skb(struct sock *sk, struct
       sk_buff *skb)
67  {
68 - skb_truesize_check(skb);
69 + /* Modified @ Simula
70 +    skb_truesize_check creates unnecessary
71 +    noise when combined with RDB */
72 + //skb_truesize_check(skb);
73    sock_set_flag(sk, SOCK_QUEUE_SHRUNK);
74    sk->sk_wmem_queued   -= skb->truesize;
75    sk->sk_forward_alloc += skb->truesize;
76 diff --git a/include/net/tcp.h b/include/net/tcp.h
77 index 54053de..411cc9b 100644
78 --- a/include/net/tcp.h
79 +++ b/include/net/tcp.h
80 @@ -188,9 +188,19 @@ extern void tcp_time_wait(struct sock *sk,
       int state, int timeo);
81  #define TCP_NAGLE_CORK   2 /* Socket is corked     */
82  #define TCP_NAGLE_PUSH   4 /* Cork is overridden for already
       queued data */
83
84 +/* Added @ Simula - Thin stream support */
85 +#define TCP_FORCE_THIN_RDB            0 /* Thin streams: exp.
       backoff   default off */
86 +#define TCP_FORCE_THIN_RM_EXPB        0 /* Thin streams:
       dynamic dupack default off */
87 +#define TCP_FORCE_THIN_DUPACK         0 /* Thin streams:
       smaller minRTO default off */
```

```
88  +#define TCP_RDB_MAX_BUNDLE_BYTES      0 /* Thin streams: Limit
        maximum bundled bytes */
89  +
90   extern struct inet_timewait_death_row tcp_death_row;
91
92   /* sysctl variables for tcp */
93  +extern int sysctl_tcp_force_thin_rdb;         /* Added @
        Simula */
94  +extern int sysctl_tcp_force_thin_rm_expb;     /* Added @
        Simula */
95  +extern int sysctl_tcp_force_thin_dupack;      /* Added @
        Simula */
96  +extern int sysctl_tcp_rdb_max_bundle_bytes;   /* Added @
        Simula */
97   extern int sysctl_tcp_timestamps;
98   extern int sysctl_tcp_window_scaling;
99   extern int sysctl_tcp_sack;
100 @@ -723,6 +733,16 @@ static inline unsigned int
        tcp_packets_in_flight(const struct tcp_sock *tp)
101   return (tp->packets_out - tp->left_out + tp->retrans_out);
102  }
103
104 +/* Added @ Simula
105 + *
106 + * To determine whether a stream is thin or not
107 + * return 1 if thin, 0 othervice
108 + */
109 +static inline unsigned int tcp_stream_is_thin(const struct
        tcp_sock *tp)
110 +{
111 + return (tp->packets_out < 4 ? 1 : 0);
112 +}
113 +
114  /* If cwnd > ssthresh, we may raise ssthresh to be half-way to
        cwnd.
115   * The exception is rate halving phase, when cwnd is
        decreasing towards
116   * ssthresh.
117  diff --git a/net/ipv4/sysctl_net_ipv4.c b/net/ipv4/
        sysctl_net_ipv4.c
118  index 53ef0f4..58ac82b 100644
119  --- a/net/ipv4/sysctl_net_ipv4.c
120  +++ b/net/ipv4/sysctl_net_ipv4.c
121  @@ -187,6 +187,38 @@ static int
        strategy_allowed_congestion_control(ctl_table *table, int
        __user *nam
122   }
123
124   ctl_table ipv4_table[] = {
125  + {       /* Added @ Simula for thin streams */
126  +                .ctl_name       = NET_IPV4_TCP_FORCE_THIN_RDB,
127  +                .procname       = "tcp_force_thin_rdb",
128  +                .data           = &sysctl_tcp_force_thin_rdb,
129  +                .maxlen         = sizeof(int),
130  +                .mode           = 0644,
131  +                .proc_handler   = &proc_dointvec
132  +        },
133  + {       /* Added @ Simula for thin streams */
134  +                .ctl_name       =
        NET_IPV4_TCP_FORCE_THIN_RM_EXPB,
135  +                .procname       = "tcp_force_thin_rm_expb",
136  +                .data           = &
        sysctl_tcp_force_thin_rm_expb,
137  +                .maxlen         = sizeof(int),
138  +                .mode           = 0644,
```

```
139 +                .proc_handler   = &proc_dointvec
140 +        },
141 + {       /* Added @ Simula for thin streams */
142 +                .ctl_name       =
        NET_IPV4_TCP_FORCE_THIN_DUPACK,
143 +                .procname       = "tcp_force_thin_dupack",
144 +                .data           = &
        sysctl_tcp_force_thin_dupack,
145 +                .maxlen         = sizeof(int),
146 +                .mode           = 0644,
147 +                .proc_handler   = &proc_dointvec
148 +        },
149 + {       /* Added @ Simula for thin streams */
150 +                .ctl_name       =
        NET_IPV4_TCP_RDB_MAX_BUNDLE_BYTES,
151 +                .procname       = "tcp_rdb_max_bundle_bytes",
152 +                .data           = &
        sysctl_tcp_rdb_max_bundle_bytes,
153 +                .maxlen         = sizeof(int),
154 +                .mode           = 0644,
155 +                .proc_handler   = &proc_dointvec
156 +        },
157   {
158     .ctl_name = NET_IPV4_TCP_TIMESTAMPS,
159     .procname = "tcp_timestamps",
160 diff --git a/net/ipv4/tcp.c b/net/ipv4/tcp.c
161 index 7e74011..8aeec1b 100644
162 --- a/net/ipv4/tcp.c
163 +++ b/net/ipv4/tcp.c
164 @@ -270,6 +270,10 @@
165
166  int sysctl_tcp_fin_timeout __read_mostly = TCP_FIN_TIMEOUT;
167
168 +/* Added @ Simula */
169 +int sysctl_tcp_force_thin_rdb __read_mostly =
        TCP_FORCE_THIN_RDB;
170 +int sysctl_tcp_rdb_max_bundle_bytes __read_mostly =
        TCP_RDB_MAX_BUNDLE_BYTES;
171 +
172  DEFINE_SNMP_STAT(struct tcp_mib, tcp_statistics) __read_mostly
        ;
173
174  atomic_t tcp_orphan_count = ATOMIC_INIT(0);
175 @@ -658,6 +662,167 @@ static inline int select_size(struct sock
        *sk)
176   return tmp;
177  }
178
179 +/* Added at Simula to support RDB */
180 +static int tcp_trans_merge_prev(struct sock *sk, struct
        sk_buff *skb, int mss_now)
181 +{
182 + struct tcp_sock *tp = tcp_sk(sk);
183 +
184 + /* Make sure that this isn't referenced by somebody else */
185 +
186 + if(!skb_cloned(skb)){
187 +    struct sk_buff *prev_skb = skb->prev;
188 +    int skb_size = skb->len;
189 +    int old_headlen = 0;
190 +    int ua_data = 0;
191 +    int uad_head = 0;
192 +    int uad_frags = 0;
193 +    int ua_nr_frags = 0;
194 +    int ua_frags_diff = 0;
```

```
195 +
196 +   /* Since this technique currently does not support SACK, I
197 +    * return -1 if the previous has been SACK'd. */
198 +   if(TCP_SKB_CB(prev_skb)->sacked & TCPCB_SACKED_ACKED){
199 +     return -1;
200 +   }
201 +
202 +   /* Current skb is out of window. */
203 +   if (after(TCP_SKB_CB(skb)->end_seq, tp->snd_una+tp->snd_wnd
    )){
204 +     return -1;
205 +   }
206 +
207 +   /*TODO: Optimize this part with regards to how the
208 +     variables are initialized */
209 +
210 +   /*Calculates the ammount of unacked data that is available
    */
211 +   ua_data = (TCP_SKB_CB(prev_skb)->end_seq - tp->snd_una >
212 +        prev_skb->len ? prev_skb->len :
213 +        TCP_SKB_CB(prev_skb)->end_seq - tp->snd_una);
214 +   ua_frags_diff = ua_data - prev_skb->data_len;
215 +   uad_frags = (ua_frags_diff > 0 ? prev_skb->data_len :
    ua_data);
216 +   uad_head = (ua_frags_diff > 0 ? ua_data - uad_frags : 0);
217 +
218 +   if(ua_data <= 0)
219 +     return -1;
220 +
221 +   if(uad_frags > 0){
222 +     int i = 0;
223 +     int bytes_frags = 0;
224 +
225 +     if(uad_frags == prev_skb->data_len){
226 +       ua_nr_frags = skb_shinfo(prev_skb)->nr_frags;
227 +     } else{
228 +       for(i=skb_shinfo(prev_skb)->nr_frags - 1; i>=0; i--){
229 +         if(skb_shinfo(prev_skb)->frags[i].size
230 +            + bytes_frags == uad_frags){
231 +           ua_nr_frags += 1;
232 +           break;
233 +         }
234 +         ua_nr_frags += 1;
235 +         bytes_frags += skb_shinfo(prev_skb)->frags[i].size;
236 +       }
237 +     }
238 +   }
239 +
240 +   /*
241 +    * Do the diffrenet checks on size and content, and return
    if
242 +    * something will not work.
243 +    *
244 +    * TODO: Support copying some bytes
245 +    *
246 +    * 1. Larger than MSS.
247 +    * 2. Enough room for the stuff stored in the linear area
248 +    * 3. Enoug room for the pages
249 +    * 4. If both skbs have some data stored in the linear area
    , and prev_skb
250 +    * also has some stored in the paged area, they cannot be
    merged easily.
251 +    * 5. If prev_skb is linear, then this one has to be it as
    well.
252 +    */
```

```
253 +   if ((sysctl_tcp_rdb_max_bundle_bytes == 0 && ((skb_size +
        ua_data) > mss_now))
254 +       || (sysctl_tcp_rdb_max_bundle_bytes > 0 && ((skb_size +
         ua_data) >
255 +               sysctl_tcp_rdb_max_bundle_bytes))){
256 +     return -1;
257 +   }
258 +
259 +   /* We need to know tailroom, even if it is nonlinear */
260 +   if(uad_head > (skb->end - skb->tail)){
261 +     return -1;
262 +   }
263 +
264 +   if(skb_is_nonlinear(skb) && (uad_frags > 0)){
265 +     if((ua_nr_frags +
266 +         skb_shinfo(skb)->nr_frags) > MAX_SKB_FRAGS){
267 +       return -1;
268 +     }
269 +
270 +     if(skb_headlen(skb) > 0){
271 +       return -1;
272 +     }
273 +   }
274 +
275 +   if((uad_frags > 0) && skb_headlen(skb) > 0){
276 +     return -1;
277 +   }
278 +
279 +   /* To avoid duplicate copies (and copies
280 +      where parts have been acked) */
281 +   if(TCP_SKB_CB(skb)->seq <= (TCP_SKB_CB(prev_skb)->end_seq -
         ua_data)){
282 +     return -1;
283 +   }
284 +
285 +   /*SYN's are holy*/
286 +   if(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_SYN || TCP_SKB_CB(
        skb)->flags & TCPCB_FLAG_FIN){
287 +     return -1;
288 +   }
289 +
290 +   /* Copy linear data */
291 +   if(uad_head > 0){
292 +
293 +     /* Add required space to the header. Can't use put due to
         linearity */
294 +     old_headlen = skb_headlen(skb);
295 +     skb->tail += uad_head;
296 +     skb->len += uad_head;
297 +
298 +     if(skb_headlen(skb) > 0){
299 +       memmove(skb->data + uad_head, skb->data, old_headlen);
300 +     }
301 +
302 +     skb_copy_to_linear_data(skb, prev_skb->data + (
        skb_headlen(prev_skb) - uad_head), uad_head);
303 +   }
304 +
305 +   /*Copy paged data*/
306 +   if(uad_frags > 0){
307 +     int i = 0;
308 +     /*Must move data backwards in the array.*/
309 +     if(skb_is_nonlinear(skb)){
310 +       memmove(skb_shinfo(skb)->frags + ua_nr_frags,
311 +         skb_shinfo(skb)->frags,
```

```
312 +          skb_shinfo(skb)->nr_frags*sizeof(skb_frag_t));
313 +      }
314 +
315 +      /*Copy info and update pages*/
316 +      memcpy(skb_shinfo(skb)->frags,
317 +             skb_shinfo(prev_skb)->frags + (skb_shinfo(prev_skb
    )->nr_frags - ua_nr_frags),
318 +             ua_nr_frags*sizeof(skb_frag_t));
319 +
320 +      for(i=0; i<ua_nr_frags;i++){
321 +        get_page(skb_shinfo(skb)->frags[i].page);
322 +      }
323 +
324 +      skb_shinfo(skb)->nr_frags += ua_nr_frags;
325 +      skb->data_len += uad_frags;
326 +      skb->len += uad_frags;
327 +    }
328 +
329 +    TCP_SKB_CB(skb)->seq = TCP_SKB_CB(prev_skb)->end_seq -
    ua_data;
330 +
331 +    if(skb->ip_summed == CHECKSUM_PARTIAL)
332 +      skb->csum = CHECKSUM_PARTIAL;
333 +    else
334 +      skb->csum = skb_checksum(skb, 0, skb->len, 0);
335 + }
336 +
337 + return 1;
338 +}
339 +
340  int tcp_sendmsg(struct kiocb *iocb, struct socket *sock,
        struct msghdr *msg,
341      size_t size)
342  {
343 @@ -825,6 +990,16 @@ new_segment:
344
345        from += copy;
346        copied += copy;
347 +
348 +      /* Added at Simula to support RDB */
349 +      if(((tp->thin_rdb || sysctl_tcp_force_thin_rdb)) && skb->
    len < mss_now){
350 +        if(skb->prev != (struct sk_buff*) &(sk)->sk_write_queue
351 +           && !(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_SYN)
352 +           && !(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_FIN)){
353 +          tcp_trans_merge_prev(sk, skb, mss_now);
354 +        }
355 +      } /* End - Simula */
356 +
357        if ((seglen -= copy) == 0 && iovlen == 0)
358          goto out;
359
360 @@ -1870,7 +2045,25 @@ static int do_tcp_setsockopt(struct sock
     *sk, int level,
361        tcp_push_pending_frames(sk);
362      }
363      break;
364 -
365 +
366 +        /* Added @ Simula. Support for thin streams */
367 + case TCP_THIN_RDB:
368 +    if (val)
369 +      tp->thin_rdb = 1;
370 +    break;
371 +
```

```
372 +        /* Added @ Simula. Support for thin streams */
373 + case TCP_THIN_RM_EXPB:
374 +   if (val)
375 +     tp->thin_rm_expb = 1;
376 +   break;
377 +
378 +        /* Added @ Simula. Support for thin streams */
379 + case TCP_THIN_DUPACK:
380 +   if (val)
381 +     tp->thin_dupack = 1;
382 +   break;
383 +
384   case TCP_KEEPIDLE:
385     if (val < 1 || val > MAX_TCP_KEEPIDLE)
386       err = -EINVAL;
387 diff --git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
388 index f893e90..f42ef14 100644
389 --- a/net/ipv4/tcp_input.c
390 +++ b/net/ipv4/tcp_input.c
391 @@ -89,6 +89,9 @@ int sysctl_tcp_frto __read_mostly;
392  int sysctl_tcp_frto_response __read_mostly;
393  int sysctl_tcp_nometrics_save __read_mostly;
394
395 +/* Added @ Simula */
396 +int sysctl_tcp_force_thin_dupack __read_mostly =
        TCP_FORCE_THIN_DUPACK;
397 +
398  int sysctl_tcp_moderate_rcvbuf __read_mostly = 1;
399  int sysctl_tcp_abc __read_mostly;
400
401 @@ -1704,6 +1707,12 @@ static int tcp_time_to_recover(struct
        sock *sk)
402       */
403      return 1;
404    }
405 +
406 + /*Added at Simula to modify fast retransmit */
407 + if ((tp->thin_dupack || sysctl_tcp_force_thin_dupack) &&
408 +     tcp_fackets_out(tp) > 1 && tcp_stream_is_thin(tp)){
409 +   return 1;
410 + }
411
412    return 0;
413  }
414 @@ -2437,30 +2446,127 @@ static int tcp_clean_rtx_queue(struct
        sock *sk, __s32 *seq_rtt_p)
415  {
416    struct tcp_sock *tp = tcp_sk(sk);
417    const struct inet_connection_sock *icsk = inet_csk(sk);
418 - struct sk_buff *skb;
419 + struct sk_buff *skb = tcp_write_queue_head(sk);
420 + struct sk_buff *next_skb;
421 +
422    __u32 now = tcp_time_stamp;
423    int acked = 0;
424    int prior_packets = tp->packets_out;
425 +
426 + /*Added at Simula for RDB support*/
427 + __u8 done = 0;
428 + int remove = 0;
429 + int remove_head = 0;
430 + int remove_frags = 0;
431 + int no_frags;
432 + int data_frags;
433 + int i;
```

```
434 +
435   __s32 seq_rtt = -1;
436   ktime_t last_ackt = net_invalid_timestamp();
437 -
438 - while ((skb = tcp_write_queue_head(sk)) &&
439 -        skb != tcp_send_head(sk)) {
440 +
441 + while (skb != NULL
442 +        && ((!(tp->thin_rdb || sysctl_tcp_force_thin_rdb)
443 +       && skb != tcp_send_head(sk)
444 +       && skb != (struct sk_buff *)&sk->sk_write_queue)
445 +      || ((tp->thin_rdb || sysctl_tcp_force_thin_rdb)
446 +          && skb != (struct sk_buff *)&sk->sk_write_queue))){
447     struct tcp_skb_cb *scb = TCP_SKB_CB(skb);
448     __u8 sacked = scb->sacked;
449 -
450 +
451 +   if(skb == NULL){
452 +     break;
453 +   }
454 +
455 +   if(skb == tcp_send_head(sk)){
456 +     break;
457 +   }
458 +
459 +   if(skb == (struct sk_buff *)&sk->sk_write_queue){
460 +     break;
461 +   }
462 +
463     /* If our packet is before the ack sequence we can
464      * discard it as it's confirmed to have arrived at
465      * the other end.
466      */
467     if (after(scb->end_seq, tp->snd_una)) {
468 -     if (tcp_skb_pcount(skb) > 1 &&
469 -         after(tp->snd_una, scb->seq))
470 -       acked |= tcp_tso_acked(sk, skb,
471 -                   now, &seq_rtt);
472 -     break;
473 +     if (tcp_skb_pcount(skb) > 1 && after(tp->snd_una, scb->
    seq))
474 +       acked |= tcp_tso_acked(sk, skb, now, &seq_rtt);
475 +
476 +     done = 1;
477 +
478 +     /* Added at Simula for RDB support*/
479 +     if ((tp->thin_rdb || sysctl_tcp_force_thin_rdb) && after(
    tp->snd_una, scb->seq)) {
480 +       if (!skb_cloned(skb) && !(scb->flags & TCPCB_FLAG_SYN))
    {
481 +         remove = tp->snd_una - scb->seq;
482 +         remove_head = (remove > skb_headlen(skb) ?
483 +                   skb_headlen(skb) : remove);
484 +         remove_frags = (remove > skb_headlen(skb) ?
485 +             remove - remove_head : 0);
486 +
487 +         /* Has linear data */
488 +         if(skb_headlen(skb) > 0 && remove_head > 0){
489 +           memmove(skb->data,
490 +             skb->data + remove_head,
491 +             skb_headlen(skb) - remove_head);
492 +
493 +           skb->tail -= remove_head;
494 +         }
495 +
```

```
496 +          if(skb_is_nonlinear(skb) && remove_frags > 0){
497 +            no_frags = 0;
498 +            data_frags = 0;
499 +
500 +            /*Remove unecessary pages*/
501 +            for(i=0; i<skb_shinfo(skb)->nr_frags; i++){
502 +              if(data_frags + skb_shinfo(skb)->frags[i].size
503 +                 == remove_frags){
504 +                put_page(skb_shinfo(skb)->frags[i].page);
505 +                no_frags += 1;
506 +                break;
507 +              }
508 +              put_page(skb_shinfo(skb)->frags[i].page);
509 +              no_frags += 1;
510 +              data_frags += skb_shinfo(skb)->frags[i].size;
511 +            }
512 +
513 +            if(skb_shinfo(skb)->nr_frags > no_frags)
514 +              memmove(skb_shinfo(skb)->frags,
515 +                skb_shinfo(skb)->frags + no_frags,
516 +                (skb_shinfo(skb)->nr_frags
517 +                 - no_frags)*sizeof(skb_frag_t));
518 +
519 +            skb->data_len -= remove_frags;
520 +            skb_shinfo(skb)->nr_frags -= no_frags;
521 +
522 +          }
523 +
524 +          scb->seq += remove;
525 +          skb->len -= remove;
526 +
527 +          if(skb->ip_summed == CHECKSUM_PARTIAL)
528 +            skb->csum = CHECKSUM_PARTIAL;
529 +          else
530 +            skb->csum = skb_checksum(skb, 0, skb->len, 0);
531 +
532 +        }
533 +
534 +        /*Only move forward if data could be removed from this
    packet*/
535 +        done = 2;
536 +
537 +      }
538 +
539 +      if(done == 1 || tcp_skb_is_last(sk,skb)){
540 +        break;
541 +      } else if(done == 2){
542 +        skb = skb->next;
543 +        done = 1;
544 +        continue;
545 +      }
546 +
547       }
548 -
549 +
550       /* Initial outgoing SYN's get put onto the write_queue
551        * just like anything else we transmit.  It is not
552        * true data, and if we misinform our callers that
553 @@ -2474,14 +2580,14 @@ static int tcp_clean_rtx_queue(struct
    sock *sk, __s32 *seq_rtt_p)
554         acked |= FLAG_SYN_ACKED;
555         tp->retrans_stamp = 0;
556       }
557 -
558 +
```

```
559       /* MTU probing checks */
560       if (icsk->icsk_mtup.probe_size) {
561         if (!after(tp->mtu_probe.probe_seq_end, TCP_SKB_CB(skb)->
              end_seq)) {
562           tcp_mtup_probe_success(sk, skb);
563         }
564       }
565  -
566  +
567       if (sacked) {
568         if (sacked & TCPCB_RETRANS) {
569           if (sacked & TCPCB_SACKED_RETRANS)
570  @@ -2505,24 +2611,32 @@ static int tcp_clean_rtx_queue(struct
         sock *sk, __s32 *seq_rtt_p)
571         seq_rtt = now - scb->when;
572         last_ackt = skb->tstamp;
573       }
574  +
575  +    if ((tp->thin_rdb || sysctl_tcp_force_thin_rdb) && skb ==
         tcp_send_head(sk)) {
576  +      tcp_advance_send_head(sk, skb);
577  +    }
578  +
579       tcp_dec_pcount_approx(&tp->fackets_out, skb);
580       tcp_packets_out_dec(tp, skb);
581  +    next_skb = skb->next;
582       tcp_unlink_write_queue(skb, sk);
583       sk_stream_free_skb(sk, skb);
584       clear_all_retrans_hints(tp);
585  +    /* Added at Simula to support RDB */
586  +    skb = next_skb;
587     }
588  -
589  +
590     if (acked&FLAG_ACKED) {
591       u32 pkts_acked = prior_packets - tp->packets_out;
592       const struct tcp_congestion_ops *ca_ops
593         = inet_csk(sk)->icsk_ca_ops;
594  -
595  +
596       tcp_ack_update_rtt(sk, acked, seq_rtt);
597       tcp_ack_packets_out(sk);
598  -
599  +
600       if (ca_ops->pkts_acked) {
601         s32 rtt_us = -1;
602  -
603  +
604         /* Is the ACK triggering packet unambiguous? */
605         if (!(acked & FLAG_RETRANS_DATA_ACKED)) {
606           /* High resolution needed and available? */
607  diff --git a/net/ipv4/tcp_output.c b/net/ipv4/tcp_output.c
608  index 666d8a5..daa580d 100644
609  --- a/net/ipv4/tcp_output.c
610  +++ b/net/ipv4/tcp_output.c
611  @@ -1653,7 +1653,7 @@ static void tcp_retrans_try_collapse(
         struct sock *sk, struct sk_buff *skb, int m
612
613       BUG_ON(tcp_skb_pcount(skb) != 1 ||
614              tcp_skb_pcount(next_skb) != 1);
615  -
616  +
617       /* changing transmit queue under us so clear hints */
618       clear_all_retrans_hints(tp);
619
```

```
@@ -1702,6 +1702,166 @@ static void tcp_retrans_try_collapse(
     struct sock *sk, struct sk_buff *skb, int m
   }
 }

+/* Added at Simula. Variation of the regular collapse,
+   adapted to support RDB  */
+static void tcp_retrans_merge_redundant(struct sock *sk,
+         struct sk_buff *skb,
+         int mss_now)
+{
+ struct tcp_sock *tp = tcp_sk(sk);
+ struct sk_buff *next_skb = skb->next;
+ int skb_size = skb->len;
+ int new_data = 0;
+ int new_data_head = 0;
+ int new_data_frags = 0;
+ int new_frags = 0;
+ int old_headlen = 0;
+
+ int i;
+ int data_frags = 0;
+
+ /* Loop through as many packets as possible
+  * (will create a lot of redundant data, but WHATEVER).
+  * The only packet this MIGHT be critical for is
+  * if this packet is the last in the retrans-queue.
+  *
+  * Make sure that the first skb isnt already in
+  * use by somebody else. */
+
+ if (!skb_cloned(skb)) {
+   /* Iterate through the retransmit queue */
+   for (; (next_skb != (sk)->sk_send_head) &&
+          (next_skb != (struct sk_buff *) &(sk)->
     sk_write_queue);
+        next_skb = next_skb->next) {
+
+     /* Reset variables */
+     new_frags = 0;
+     data_frags = 0;
+     new_data = TCP_SKB_CB(next_skb)->end_seq - TCP_SKB_CB(skb
    )->end_seq;
+
+     /* New data will be stored at skb->start_add +
    some_offset,
+        in other words the last N bytes */
+     new_data_frags = (new_data > next_skb->data_len ?
+          next_skb->data_len : new_data);
+     new_data_head = (new_data > next_skb->data_len ?
+          new_data - skb->data_len : 0);
+
+     /*
+      * 1. Contains the same data
+      * 2. Size
+      * 3. Sack
+      * 4. Window
+      * 5. Cannot merge with a later packet that has linear
    data
+      * 6. The new number of frags will exceed the limit
+      * 7. Enough tailroom
+      */
+
+     if(new_data <= 0){
+       return;
```

```
680 +     }
681 +
682 +     if ((sysctl_tcp_rdb_max_bundle_bytes == 0 && ((skb_size +
        new_data) > mss_now))
683 +         || (sysctl_tcp_rdb_max_bundle_bytes > 0 && ((skb_size
        + new_data) >
684 +                 sysctl_tcp_rdb_max_bundle_bytes))){
685 +       return;
686 +     }
687 +
688 +     if(TCP_SKB_CB(next_skb)->flags & TCPCB_FLAG_FIN){
689 +       return;
690 +     }
691 +
692 +     if((TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED) ||
693 +        (TCP_SKB_CB(next_skb)->sacked & TCPCB_SACKED_ACKED)){
694 +       return;
695 +     }
696 +
697 +     if(after(TCP_SKB_CB(skb)->end_seq + new_data, tp->snd_una
        + tp->snd_wnd)){
698 +       return;
699 +     }
700 +
701 +     if(skb_shinfo(skb)->frag_list || skb_shinfo(skb)->
        frag_list){
702 +       return;
703 +     }
704 +
705 +     /* Calculate number of new fragments. Any new data will
        be
706 +        stored in the back. */
707 +     if(skb_is_nonlinear(next_skb)){
708 +       i = (skb_shinfo(next_skb)->nr_frags == 0 ?
709 +            0 : skb_shinfo(next_skb)->nr_frags - 1);
710 +       for( ; i>=0;i--){
711 +         if(data_frags + skb_shinfo(next_skb)->frags[i].size
        ==
712 +            new_data_frags){
713 +           new_frags += 1;
714 +           break;
715 +         }
716 +
717 +         data_frags += skb_shinfo(next_skb)->frags[i].size;
718 +         new_frags += 1;
719 +       }
720 +     }
721 +
722 +     /* If dealing with a fragmented skb, only merge
723 +        with an skb that ONLY contain frags */
724 +     if(skb_is_nonlinear(skb)){
725 +
726 +       /*Due to the way packets are processed, no later data*/
727 +       if(skb_headlen(next_skb) && new_data_head > 0){
728 +         return;
729 +       }
730 +
731 +       if(skb_is_nonlinear(next_skb) && (new_data_frags > 0)
        &&
732 +          ((skb_shinfo(skb)->nr_frags + new_frags) >
        MAX_SKB_FRAGS)){
733 +         return;
734 +       }
735 +
736 +     } else {
```

```
737 +       if(skb_headlen(next_skb) && (new_data_head > (skb->end
    - skb->tail))){
738 +         return;
739 +       }
740 +     }
741 +
742 +     /*Copy linear data. This will only occur if both are
    linear,
743 +       or only A is linear*/
744 +     if(skb_headlen(next_skb) && (new_data_head > 0)){
745 +       old_headlen = skb_headlen(skb);
746 +       skb->tail += new_data_head;
747 +       skb->len += new_data_head;
748 +
749 +       /* The new data starts in the linear area,
750 +          and the correct offset will then be given by
751 +          removing new_data ammount of bytes from length. */
752 +       skb_copy_to_linear_data_offset(skb, old_headlen,
    next_skb->tail -
753 +                      new_data_head, new_data_head);
754 +     }
755 +
756 +     if(skb_is_nonlinear(next_skb) && (new_data_frags > 0)){
757 +       memcpy(skb_shinfo(skb)->frags + skb_shinfo(skb)->
    nr_frags,
758 +              skb_shinfo(next_skb)->frags +
759 +              (skb_shinfo(next_skb)->nr_frags - new_frags),
760 +              new_frags*sizeof(skb_frag_t));
761 +
762 +       for(i=skb_shinfo(skb)->nr_frags;
763 +           i < skb_shinfo(skb)->nr_frags + new_frags; i++)
764 +         get_page(skb_shinfo(skb)->frags[i].page);
765 +
766 +       skb_shinfo(skb)->nr_frags += new_frags;
767 +       skb->data_len += new_data_frags;
768 +       skb->len += new_data_frags;
769 +     }
770 +
771 +     TCP_SKB_CB(skb)->end_seq += new_data;
772 +
773 +     if(skb->ip_summed == CHECKSUM_PARTIAL)
774 +       skb->csum = CHECKSUM_PARTIAL;
775 +     else
776 +       skb->csum = skb_checksum(skb, 0, skb->len, 0);
777 +
778 +     skb_size = skb->len;
779 +   }
780 +
781 + }
782 +}
783 +
784   /* Do a simple retransmit without using the backoff mechanisms
        in
785    * tcp_timer. This is used for path mtu discovery.
786    * The socket is already locked here.
787  @@ -1756,6 +1916,8 @@ void tcp_simple_retransmit(struct sock *
       sk)
788   /* This retransmits one SKB.  Policy decisions and retransmit
        queue
789    * state updates are done by the caller.  Returns non-zero if
        an
790    * error occurred which prevented the send.
791  + * Modified at Simula to support thin stream optimizations
792  + * TODO: Update to use new helpers (like tcp_write_queue_next
       ())
```

```
793    */
794   int tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
795   {
796  @@ -1802,10 +1964,21 @@ int tcp_retransmit_skb(struct sock *sk,
       struct sk_buff *skb)
797         (skb->len < (cur_mss >> 1)) &&
798         (tcp_write_queue_next(sk, skb) != tcp_send_head(sk)) &&
799         (!tcp_skb_is_last(sk, skb)) &&
800  -      (skb_shinfo(skb)->nr_frags == 0 && skb_shinfo(
       tcp_write_queue_next(sk, skb))->nr_frags == 0) &&
801  -      (tcp_skb_pcount(skb) == 1 && tcp_skb_pcount(
       tcp_write_queue_next(sk, skb)) == 1) &&
802  -      (sysctl_tcp_retrans_collapse != 0))
803  +      (skb_shinfo(skb)->nr_frags == 0
804  +       && skb_shinfo(tcp_write_queue_next(sk, skb))->nr_frags
       == 0)
805  +      && (tcp_skb_pcount(skb) == 1
806  +     && tcp_skb_pcount(tcp_write_queue_next(sk, skb)) == 1)
807  +      && (sysctl_tcp_retrans_collapse != 0)
808  +      && !((tp->thin_rdb || sysctl_tcp_force_thin_rdb))) {
809       tcp_retrans_try_collapse(sk, skb, cur_mss);
810  +         } else if ((tp->thin_rdb || sysctl_tcp_force_thin_rdb
       )) {
811  +     if (!(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_SYN) &&
812  +         !(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_FIN) &&
813  +         (skb->next != tcp_send_head(sk)) &&
814  +         (skb->next != (struct sk_buff *) &sk->sk_write_queue))
       {
815  +       tcp_retrans_merge_redundant(sk, skb, cur_mss);
816  +     }
817  + }
818
819     if (inet_csk(sk)->icsk_af_ops->rebuild_header(sk))
820       return -EHOSTUNREACH; /* Routing failure or similar. */
821   diff --git a/net/ipv4/tcp_timer.c b/net/ipv4/tcp_timer.c
822   index e9b151b..ad8de35 100644
823   --- a/net/ipv4/tcp_timer.c
824   +++ b/net/ipv4/tcp_timer.c
825   @@ -32,6 +32,9 @@ int sysctl_tcp_retries1 __read_mostly =
       TCP_RETR1;
826    int sysctl_tcp_retries2 __read_mostly = TCP_RETR2;
827    int sysctl_tcp_orphan_retries __read_mostly;
828
829   +/* Added @ Simula */
830   +int sysctl_tcp_force_thin_rm_expb __read_mostly =
       TCP_FORCE_THIN_RM_EXPB;
831   +
832    static void tcp_write_timer(unsigned long);
833    static void tcp_delack_timer(unsigned long);
834    static void tcp_keepalive_timer (unsigned long data);
835   @@ -368,13 +371,28 @@ static void tcp_retransmit_timer(struct
       sock *sk)
836       */
837     icsk->icsk_backoff++;
838     icsk->icsk_retransmits++;
839   -
840   +
841    out_reset_timer:
842   - icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
843   + /* Added @ Simula removal of exponential backoff for thin
       streams
844   +    We only want to apply this for an established stream */
845   + if ((tp->thin_rm_expb || sysctl_tcp_force_thin_rm_expb)
846   +     && tcp_stream_is_thin(tp) && sk->sk_state ==
       TCP_ESTABLISHED) {
```

```
847 +   /* Since 'icsk_backoff' is used to reset timer, set to 0
848 +    * Recalculate 'icsk_rto' as this might be increased if
        stream oscillates
849 +    * between thin and thick, thus the old value might already
         be too high
850 +    * compared to the value set by 'tcp_set_rto' in tcp_input.
        c which resets
851 +    * the rto without backoff. */
852 +   icsk->icsk_backoff = 0;
853 +   icsk->icsk_rto = min(((tp->srtt >> 3) + tp->rttvar),
        TCP_RTO_MAX);
854 + } else {
855 +   /* Use normal backoff */
856 +   icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
857 + }
858 + /* End Simula*/
859   inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS, icsk->
         icsk_rto, TCP_RTO_MAX);
860   if (icsk->icsk_retransmits > sysctl_tcp_retries1)
861     __sk_dst_reset(sk);
862 -
863 +
864  out:;
865  }
866
867 --
868 1.5.6.3
```

This is the initial patch sent to Linux devs containing all the three thin-stream modifications.

Code Listing C.1: Patch for RDBv1, mFR and LT for Linux kernel 2.6.23

C.2 Netem with fixed loss

Source Code

```
diff --git a/include/uapi/linux/pkt_sched.h b/include/uapi/
    linux/pkt_sched.h
index 9b82913..2ebe95b 100644
--- a/include/uapi/linux/pkt_sched.h
+++ b/include/uapi/linux/pkt_sched.h
@@ -560,6 +560,7 @@ enum {
  NETEM_LOSS_UNSPEC,
  NETEM_LOSS_GI,    /* General Intuitive - 4 state model */
  NETEM_LOSS_GE,    /* Gilbert Elliot models */
+ NETEM_LOSS_FIXED, /* Lose fixed packets */
  __NETEM_LOSS_MAX
 };
 #define NETEM_LOSS_MAX (__NETEM_LOSS_MAX - 1)
@@ -581,6 +582,16 @@ struct tc_netem_gemodel {
  __u32 k1;
 };

+/* Fixed loss model */
+struct tc_netem_fixedmodel {
+ __u32 loss[50];
+ __u32 loss_length;
+ __u32 flow_length;
+ __u32 packets_processed;
+ __u32 packets_dropped;
+ __u32 verbose;
+};
+
 #define NETEM_DIST_SCALE 8192
 #define NETEM_DIST_MAX   16384

diff --git a/net/sched/sch_netem.c b/net/sched/sch_netem.c
index b87e83d..ceeda7ab 100644
--- a/net/sched/sch_netem.c
+++ b/net/sched/sch_netem.c
@@ -29,6 +29,9 @@
 #include <net/pkt_sched.h>
 #include <net/inet_ecn.h>

+#include <linux/ftrace.h>
+#include <net/tcp.h>
+
 #define VERSION "1.3"

 /* Network Emulation Queuing algorithm.
@@ -84,6 +87,7 @@ struct netem_sched_data {
  u32 ecn;
  u32 limit;
  u32 counter;
+ u32 counter_fixed;
  u32 gap;
  u32 duplicate;
  u32 reorder;
@@ -108,6 +112,7 @@ struct netem_sched_data {
     CLG_RANDOM,
     CLG_4_STATES,
     CLG_GILB_ELL,
+    CLG_FIXED,
  } loss_model;

```

```
59    /* Correlated Loss Generation models */
60  @@ -123,6 +128,16 @@ struct netem_sched_data {
61       u32 a5; /* p23 used only in 4-states */
62    } clg;
63
64  + struct fixed_loss_model {
65  +    __u32 loss_length;     /* The number of packets to drop */
66  +    __u32 loss[50];        /* Array containing which packets to
         drop */
67  +    __u32 loss_index;      /* Current index in loss */
68  +    __u32 flow_length;     /* Length of the packet flow to work
         on */
69  +    __u32 flow_accounting; /* Counts from 0 to flow_length */
70  +    __u32 packets_processed;
71  +    __u32 packets_dropped;
72  +    __u32 verbose;
73  + } flm;
74   };
75
76   /* Time stamp put into socket buffer control block
77  @@ -279,6 +294,33 @@ static bool loss_gilb_ell(struct
       netem_sched_data *q)
78    return false;
79   }
80
81  +static bool loss_fixed(struct netem_sched_data *q)
82  +{
83  + bool drop = false;
84  + struct fixed_loss_model *flm = &q->flm;
85  +
86  + flm->packets_processed++;
87  +
88  + // Advance the flow accounting (which packet in the flow)
89  + if (++(flm->flow_accounting) == flm->flow_length)
90  +   flm->flow_accounting = 0;
91  +
92  + // Drop packet
93  + if (flm->flow_accounting == flm->loss[flm->loss_index]) {
94  +   drop = true;
95  +   flm->packets_dropped++;
96  +   if (++(flm->loss_index) == flm->loss_length)
97  +     flm->loss_index = 0;
98  +
99  +   if (flm->verbose) {
100 +     printk(KERN_INFO "NETEM: Dropping packet %d, next to drop
        :%d (index:%d) (Total: %d)\n", flm->flow_accounting,
101 +            flm->loss[flm->loss_index], flm->loss_index, flm->
        packets_dropped);
102 +   }
103 + }
104 + return drop;
105 +}
106 +
107 +
108  static bool loss_event(struct netem_sched_data *q)
109  {
110   switch (q->loss_model) {
111 @@ -301,6 +343,9 @@ static bool loss_event(struct
       netem_sched_data *q)
112      * the kernel logs
113      */
114     return loss_gilb_ell(q);
115 + case CLG_FIXED:
116 +   /* Drop Nth packet */
117 +   return loss_fixed(q);
```

```
118    }
119
120    return false; /* not reached */
121  @@ -406,6 +451,7 @@ static int netem_enqueue(struct sk_buff *
       skb, struct Qdisc *sch)
122    /* We don't fill cb now as skb_unshare() may invalidate it */
123    struct netem_skb_cb *cb;
124    struct sk_buff *skb2;
125  + struct fixed_loss_model *flm = &q->flm;
126    int count = 1;
127
128    /* Random duplication */
129  @@ -414,13 +460,30 @@ static int netem_enqueue(struct sk_buff *
       skb, struct Qdisc *sch)
130
131    /* Drop packet? */
132    if (loss_event(q)) {
133  -   if (q->ecn && INET_ECN_set_ce(skb))
134  +   if (q->ecn && INET_ECN_set_ce(skb)) {
135        sch->qstats.drops++; /* mark packet */
136  -   else
137  +     if (flm->verbose) {
138  +       printk(KERN_INFO "NETEM: Setting ecn and increased
       qstats.drops to %d\n", sch->qstats.drops);
139  +     }
140  +   }
141  +   else {
142        --count;
143  +   }
144    }
145  +
146    if (count == 0) {
147      sch->qstats.drops++;
148  +
149  +   if (flm->verbose) {
150  +     unsigned int hash = 0;
151  +     if (skb->sk) {
152  +       hash = (skb->sk)->sk_hash;
153  +     }
154  +     printk(KERN_INFO "NETEM: Drop: HASH: %u, SEQ: %u ENDSEQ:
       %u WHEN: %u ACK_SEQ: %u\n", hash,
155  +            TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq,
156  +            TCP_SKB_CB(skb)->when, TCP_SKB_CB(skb)->ack_seq);
157  +   }
158  +
159      kfree_skb(skb);
160      return NET_XMIT_SUCCESS | __NET_XMIT_BYPASS;
161    }
162  @@ -501,6 +564,7 @@ static int netem_enqueue(struct sk_buff *
       skb, struct Qdisc *sch)
163      cb->time_to_send = now + delay;
164      cb->tstamp_save = skb->tstamp;
165      ++q->counter;
166  +   ++q->counter_fixed;
167      tfifo_enqueue(skb, sch);
168    } else {
169      /*
170  @@ -766,6 +830,28 @@ static int get_loss_clg(struct Qdisc *sch,
       const struct nlattr *attr)
171        break;
172      }
173
174  +   case NETEM_LOSS_FIXED: {
175  +     const struct tc_netem_fixedmodel *m = nla_data(la);
176  +     int i = 0;
```

```
177 +     q->loss_model = CLG_FIXED;
178 +
179 +     q->flm.loss_length  = m->loss_length;
180 +     q->flm.flow_length = m->flow_length;
181 +     q->flm.loss_index = 0;
182 +     q->flm.flow_accounting = 0;
183 +     q->flm.packets_processed = 0;
184 +     q->flm.verbose = m->verbose;
185 +
186 +     printk(KERN_INFO "NETEM: Setting netem loss to fixed rate
        : ");
187 +     while (i < q->flm.loss_length) {
188 +       q->flm.loss[i] = m->loss[i];
189 +       printk(" %u", q->flm.loss[i]);
190 +       i++;
191 +     }
192 +     printk(KERN_INFO "NETEM: Flow length: %d\n", q->flm.
        flow_length);
193 +     printk(KERN_INFO "NETEM: Verbose level: %d\n", q->flm.
        verbose);
194 +     break;
195 +   }
196     default:
197       pr_info("netem: unknown loss type %u\n", type);
198       return -EINVAL;
199 @@ -825,6 +911,7 @@ static int netem_change(struct Qdisc *sch,
        struct nlattr *opt)
200   q->limit = qopt->limit;
201   q->gap = qopt->gap;
202   q->counter = 0;
203 + q->counter_fixed = 0;
204   q->loss = qopt->loss;
205   q->duplicate = qopt->duplicate;
206
207 @@ -929,6 +1016,23 @@ static int dump_loss_model(const struct
        netem_sched_data *q,
208       goto nla_put_failure;
209     break;
210   }
211 + case CLG_FIXED: {
212 +   struct tc_netem_fixedmodel fm = {
213 +     .loss_length = q->flm.loss_length,
214 +     .flow_length = q->flm.flow_length,
215 +     .packets_processed = q->flm.packets_processed,
216 +     .packets_dropped = q->flm.packets_dropped,
217 +     .verbose = q->flm.verbose,
218 +   };
219 +   int i;
220 +   for (i = 0; i < q->flm.loss_length; i++) {
221 +     fm.loss[i] = q->flm.loss[i];
222 +   }
223 +
224 +   if (nla_put(skb, NETEM_LOSS_FIXED, sizeof(fm), &fm))
225 +     goto nla_put_failure;
226 +   break;
227 + }
228   }
229
230   nla_nest_end(skb, nest);
```

Code Listing C.2: Netem fixed loss patch

Source Code

```
diff --git a/include/linux/pkt_sched.h b/include/linux/pkt_sched.h
index 0d5b793..c9eb5c1 100644
--- a/include/linux/pkt_sched.h
+++ b/include/linux/pkt_sched.h
@@ -529,6 +529,7 @@ enum {
  NETEM_LOSS_UNSPEC,
  NETEM_LOSS_GI,    /* General Intuitive - 4 state model */
  NETEM_LOSS_GE,    /* Gilbert Elliot models */
+ NETEM_LOSS_FIXED,   /* Lose every Nth */
  __NETEM_LOSS_MAX
 };
 #define NETEM_LOSS_MAX (__NETEM_LOSS_MAX - 1)
@@ -550,6 +551,16 @@ struct tc_netem_gemodel {
  __u32 k1;
 };

+/* Fixed loss model */
+struct tc_netem_fixedmodel {
+ __u32 loss[50];
+ __u32 loss_length;  /* Number of specified packets to lose */
+ __u32 flow_length;
+ __u32 packets_processed;
+ __u32 packets_dropped;
+ __u32 verbose;
+};
+
 #define NETEM_DIST_SCALE 8192
 #define NETEM_DIST_MAX   16384

diff --git a/tc/Makefile b/tc/Makefile
index f523adc..ed0c258 100644
--- a/tc/Makefile
+++ b/tc/Makefile
@@ -90,6 +90,7 @@ endif
 YACC := bison
 LEX := flex
 CFLAGS += -DYY_NO_INPUT
+CFLAGS += -I/home/bendiko/master/code/net-2.6/include

 MODDESTDIR := $(DESTDIR)$(patsubst /usr%,%,$(LIBDIR))/tc

diff --git a/tc/q_netem.c b/tc/q_netem.c
index 360080c..1e3edd0 100644
--- a/tc/q_netem.c
+++ b/tc/q_netem.c
@@ -38,6 +38,7 @@ static void explain(void)
  "                  [ loss random PERCENT [CORRELATION]]\n" \
  "                  [ loss state P13 [P31 [P32 [P23 P14]]]\n" \
  "                  [ loss gemodel PERCENT [R [1-H [1-K]]]\n" \
+ "                  [ loss fixed NTH [NTH] ...\n" \
  "                  [ reorder PRECENT [CORRELATION] [ gap DISTANCE ]]\n" \
  "                  [ rate RATE [PACKETOVERHEAD] [CELLSIZE] [ CELLOVERHEAD]]\n");
 }
@@ -165,6 +166,95 @@ static int get_ticks(__u32 *ticks, const char *str)
  return 0;
 }

+int parse_fixed_loss(char *str, uint32_t *store) {
+
```

```
60  + char *endptr;
61  + long val;
62  + int range = 0;
63  + int MAX_LOSS_LENGTH = 49;
64  + uint32_t last_value = -1;
65  +
66  + int loss_list_index = 0;
67  +
68  + long next_digit(char *str, char **endptr, long *result) {
69  +   errno = 0;    /* To distinguish success/failure after call
        */
70  +   *result = strtol(str, endptr, 10);
71  +
72  +   /* Check for various possible errors */
73  +   if ((errno == ERANGE && (*result == LONG_MAX || *result ==
        LONG_MIN))
74  +       || (errno != 0 && *result == 0)) {
75  +     return -1;
76  +   }
77  +
78  +   // No digits found
79  +   if (*endptr == str) {
80  +     return 0;
81  +   }
82  +   return 1;
83  + }
84  +
85  + while (1) {
86  +   int ret = next_digit(str, &endptr, &val);
87  +   if (ret == -1) {
88  +     fprintf(stderr, "Failed to parse '%s'\n", str);
89  +     perror("strtol");
90  +     return -1;
91  +   }
92  +   else if (ret == 0) {
93  +     fprintf(stderr, "No digits found in '%s'\n", str);
94  +     break;
95  +   }
96  +
97  +   if (loss_list_index > 0 && last_value >= val) {
98  +     fprintf(stderr, "Values must be in increasing order. %ld
        is smaller than %d\n", val, last_value);
99  +     exit(0);
100 +   }
101 +
102 +   // Insert range
103 +   if (range) {
104 +     while (last_value < val) {
105 +       if (loss_list_index == MAX_LOSS_LENGTH) {
106 +         fprintf(stderr, "Too many fixed losses specified! Max
         is %d\n", MAX_LOSS_LENGTH);
107 +         exit(0);
108 +       }
109 +       last_value = last_value + 1;
110 +       if (store)
111 +         store[loss_list_index] = last_value;
112 +       loss_list_index++;
113 +     }
114 +   }
115 +   else {
116 +     if (loss_list_index == MAX_LOSS_LENGTH) {
117 +       fprintf(stderr, "Too many fixed losses specified! Max
        is %d\n", MAX_LOSS_LENGTH);
118 +       exit(0);
119 +     }
```

```
120 +     last_value = val;
121 +     if (store)
122 +       store[loss_list_index] = val;
123 +     loss_list_index++;
124 +   }
125 +
126 +   // End of string
127 +   if (*endptr == '\0')
128 +     break;
129 +
130 +   // Range
131 +   range = (*endptr == ':') ? 1 : 0;
132 +   if (*endptr != ':' && *endptr != ',') {
133 +     fprintf(stderr, "Illegal value '%c'. Values must be
        separated with ',' or ':' for range.\n", *endptr);
134 +     exit(0);
135 +   }
136 +   endptr++; // Skip ':' or ','
137 +   str = endptr;
138 + }
139 +
140 + if (store)
141 +   store[loss_list_index] = 0;
142 + // Return number of specified packets to drop
143 + return loss_list_index;
144 +}
145 +
146 +
147  static int netem_parse_opt(struct qdisc_util *qu, int argc,
         char **argv,
148           struct nlmsghdr *n)
149  {
150 @@ -176,6 +266,7 @@ static int netem_parse_opt(struct
        qdisc_util *qu, int argc, char **argv,
151   struct tc_netem_corrupt corrupt;
152   struct tc_netem_gimodel gimodel;
153   struct tc_netem_gemodel gemodel;
154 + struct tc_netem_fixedmodel fixedmodel;
155   struct tc_netem_rate rate;
156   __s16 *dist_data = NULL;
157   __u16 loss_type = NETEM_LOSS_UNSPEC;
158 @@ -321,6 +412,54 @@ static int netem_parse_opt(struct
        qdisc_util *qu, int argc, char **argv,
159           explain1("loss gemodel k");
160           return -1;
161         }
162 +     } else if (!strncmp(*argv, "fixed", 5)) {
163 +       int verbose = 0;
164 +
165 +       // Enable verbose
166 +       if (!strcmp(*argv, "fixedv")) {
167 +         verbose = 1;
168 +       }
169 +
170 +       if (!NEXT_ARG_OK()) {
171 +         explain1("'loss fixed' requires that you specify
        which packets to drop.\n");
172 +         return -1;
173 +       }
174 +       NEXT_ARG();
175 +
176 +       loss_type = NETEM_LOSS_FIXED;
177 +       fixedmodel.packets_processed = 0;
178 +       fixedmodel.packets_dropped = 0;
179 +       fixedmodel.verbose = verbose;
```

```
180 +
181 +       int count = parse_fixed_loss(*argv, NULL);
182 +
183 +       if (count <= 0) {
184 +         explain1("'loss fixed' requires that you specify
        which packets to drop.\n");
185 +         return -1;
186 +       }
187 +
188 +       __u32 *loss = malloc((count + 1) * sizeof(__u32*));
189 +       count = parse_fixed_loss(*argv, loss);
190 +       fixedmodel.loss_length = (__u32) count;
191 +       fixedmodel.flow_length = loss[count - 1];
192 +
193 +       // Copy packets to drop
194 +       int i;
195 +       for (i = 0; i < count; i++) {
196 +         fixedmodel.loss[i] = loss[i];
197 +       }
198 +       free(loss);
199 +
200 +       if (NEXT_IS_NUMBER()) {
201 +         NEXT_ARG();
202 +         fixedmodel.flow_length = (__u32) atoi(*argv);
203 +         if (fixedmodel.flow_length < loss[count - 1]) {
204 +           fprintf(stderr, "'loss fixed' flow length ('%d')
        cannot be"\
205 +             "less than last specified packet ('%d').\n",
206 +             fixedmodel.flow_length, loss[count - 1]);
207 +           return -1;
208 +         }
209 +       }
210       } else {
211         fprintf(stderr, "Unknown loss parameter: %s\n",
212           *argv);
213 @@ -470,11 +609,15 @@ static int netem_parse_opt(struct
        qdisc_util *qu, int argc, char **argv,
214       if (addattr_l(n, 1024, NETEM_LOSS_GE,
215             &gemodel, sizeof(gemodel)) < 0)
216         return -1;
217 +    } else if (loss_type == NETEM_LOSS_FIXED) {
218 +      if (addattr_l(n, 1024, NETEM_LOSS_FIXED, &fixedmodel,
        sizeof(fixedmodel)) < 0)
219 +        return -1;
220 +
221     } else {
222       fprintf(stderr, "loss in the weeds!\n");
223       return -1;
224     }
225 -
226 +
227     addattr_nest_end(n, start);
228   }
229
230 @@ -500,6 +643,8 @@ static int netem_print_opt(struct
        qdisc_util *qu, FILE *f, struct rtattr *opt)
231   const struct tc_netem_corrupt *corrupt = NULL;
232   const struct tc_netem_gimodel *gimodel = NULL;
233   const struct tc_netem_gemodel *gemodel = NULL;
234 + const struct tc_netem_fixedmodel *fixed_loss_model = NULL;
235 +
236   struct tc_netem_qopt qopt;
237   const struct tc_netem_rate *rate = NULL;
238   int len = RTA_PAYLOAD(opt) - sizeof(qopt);
```

```
239 @@ -536,13 +681,15 @@ static int netem_print_opt(struct
    qdisc_util *qu, FILE *f, struct rtattr *opt)
240     }
241     if (tb[TCA_NETEM_LOSS]) {
242       struct rtattr *lb[NETEM_LOSS_MAX + 1];
243 -
244       parse_rtattr_nested(lb, NETEM_LOSS_MAX, tb[TCA_NETEM_LOSS
          ]);
245       if (lb[NETEM_LOSS_GI])
246         gemodel = RTA_DATA(lb[NETEM_LOSS_GI]);
247       if (lb[NETEM_LOSS_GE])
248         gemodel = RTA_DATA(lb[NETEM_LOSS_GE]);
249 -   }
250 +     if (lb[NETEM_LOSS_FIXED])
251 +       fixed_loss_model = RTA_DATA(lb[NETEM_LOSS_FIXED]);
252 +   }
253 +
254     if (tb[TCA_NETEM_RATE]) {
255       if (RTA_PAYLOAD(tb[TCA_NETEM_RATE]) < sizeof(*rate))
256         return -1;
257 @@ -584,6 +731,17 @@ static int netem_print_opt(struct
    qdisc_util *qu, FILE *f, struct rtattr *opt)
258     fprintf(f, " 1-k %s", sprint_percent(gemodel->k1, b1));
259   }
260
261 + if (fixed_loss_model) {
262 +   fprintf(f, " loss fixedmodel (flow length %d)\n",
      fixed_loss_model->flow_length);
263 +   fprintf(f, " packets to lose (%d):", fixed_loss_model->
      loss_length);
264 +   int i = 0;
265 +   while (i < fixed_loss_model->loss_length) {
266 +     fprintf(f, " %u", fixed_loss_model->loss[i]);
267 +     i++;
268 +   }
269 +   fprintf(f, "\n");
270 + }
271 +
272   if (qopt.duplicate) {
273     fprintf(f, " duplicate %s",
274       sprint_percent(qopt.duplicate, b1));
```

Code Listing C.3: iproute2 fixed loss patch

Appendix D

Comments from Ilpo Järvinen on Linux mailing list

Quote D.1

"Also, this approach is extremely intrusive and adding non-linear seqno things into write queue will require _you_ to do _full audit_ over every single place to verify that seqno leaps backwards won't break anything (and you'll still probably miss some cases). I wonder if you realize how easily this kind of change manifests itself as a silent data corruption on stream level and have taken appropriate actions to validate that not a single one of scenario leads to data coming as different out as was sent in (every single byte, it's not enough to declare that application worked which could well happen with corrupted data too). TCP is very coreish and such bugs will definately hit people hard."

On the approach of modifying the SKBs in the TCP output queue. (*Järvinen* [2009, b12])

Quote D.2

"It seems very intrusive solution in general. I doubt you succeed in pulling it off as is without breaking something. To me it seems rather fragile approach to make write queue seqno backleaps you're proposing. It also leads to troubles in the truesize as you have noticed. Why not just building those redundancy containing segments at the write time in case the stream is thin, then all other parts would not have to bother about

dealing these things? Number of sysctls should be minimized, if they're to be added at all. Skb work functions should be separated from tcp layer things.
If you depend on non-changing sysctl value to select right branch, you're asking for trouble as the userspave is allowed to change it during the flow as well and even during the ack processing."

Remarks at the end of email. (*Järvinen* [2009, b12])